The Gridfour Virtual Raster Store (GVRS) File Format

Specification version 1.4

G. W. Lucas
November 2022 (updated June 2024)

Whether it be the sweeping eagle in his flight, or the open apple-blossom, the toiling work-horse, the blithe swan, the branching oak, the winding stream at its base, the drifting clouds, over all the coursing sun, form ever follows function...

> Louis H. Sullivan, “The Tall Office Building Artistically Considered”
> Lippincott’s Magazine (March, 1896): pg. 403-409

Table of Contents

Table of Figures

Table of Tables

1 Introduction

This document describes the format of the data file used by the Gridfour Virtual Raster Store (GVRS). GVRS uses a single file format for both temporary files and the long-term storage of data. This document describes version 1.4 of the GVRS format.

This document is intended for software developers who are writing code to read or write GVRS data files. It is not a formal specification. Rather, its function is to provide information in an accessible form that can be applied readily to practical implementations.

In addition to this document, a reference implementation of GVRS access routines is available at the Gridfour project site at https://github.com/gwlucastrig/gridfour. The reference API is written in Java and does not include software dependencies beyond the Java standard libraries. The code is well documented and the logic it embodies should be accessible to experienced software developers. The Java implementation is designed to be readily portable to other languages. The Gridfour site also includes a Wiki giving information on the use and design of the GVRS format at https://github.com/gwlucastrig/gridfour/wiki.

1.1 What is GVRS

The GVRS API is a software tool that uses data files to manage raster (grid) data products. It is particularly useful for grids that are too large to be stored completely in memory. For applications that require a persistent data store, or require the ability to swap data between different programs or data systems, the GVRS format provides a mechanism for the preservation and exchange of data. GVRS also supports a number of options for the lossless compression of raster data.

1.2 Objectives for the GVRS Design

The GVRS design seeks a balance between simplicity and functionality. The GVRS file format is intended to be simple enough that it should require only a reasonable effort for developers to implement code "from scratch" that can read and write a GVRS file. The design and implementation of the GVRS API focuses on maintainability and sustainability. Its goal is that a user who stores data in a GVRS file today should be able to extract that data from the file ten years from now. By keeping the GVRS design focused on a manageable set of core requirements, we hope to meet that long term goal.

1.3 Use Cases and the GVRS Design

Architect Louis H. Sullivan famously claimed that "form follows function" (Sullivan, 1896). In the case of GVRS, that notion is true. The text below provides brief review of some of the use cases that led to the creation of the GVRS file specification. Knowing something about the functional requirements for the specification may help clarify the role of the various elements described in the document that follows.

- **A Virtual Raster**: GVRS APIs provide a tool for managing raster (grid) data sets that are too large to be stored in memory. The overall grid is divided into pieces ("tiles") that can be swapped

between memory and a backing data file on an on-demand basis. The tile concept is a key element in the GVRS file design.

- **Data Authoring and Collection**: GVRS APIs support applications that create, store, and edit raster data sets. The GVRS file format is organized using data-management and indexing structures that allow the content of a raster data product to grow and change throughout the development process.
- **Fast Access by Application Code:** To foster the efficient use of gridded data, the GVRS format emphasizes simplicity and consistency. The specification avoids unnecessary complexity by focusing on a lean set of functionality and relationships between data elements.
- **Integration with Other Data Systems:** To support integration into other data systems, the GVRS format specifies reference elements that can be used as database keys or for similar applications. The format also provides developers with the ability to attach their own application-specific metadata elements to a GVRS file.
- **A Testbed for Raster Data Compression:** The GVRS system originated as a testbed for investigating raster data compression algorithms. The file format defines data compression based on two well-known compression methods (Deflate and Huffman coding) and permits application code to integrate custom techniques.

2 Definitions and Conventions

2.1 Acronyms and Abbreviations

Acronym	Meaning
ASCII	American Standard Code (for) Information Interchange
CRC	Cyclic Redundancy Code
GVRS	Gridfour Virtual Raster Store
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
RFC	Request for Comments
SI	International System (from French Système International)
UAV	Unmanned Aerial Vehicle (a “drone”)
UI	User Interface
UTF-8	Unicode Transformation Format – 8-bit
UUID	Universally Unique Identifier
UUV	Unmanned Underwater Vehicle

2.2 Definitions

Term	Description
ASCII	A single-byte (7-bit) character encoding used for most computer programming languages.
Big Endian	A scheme in which byte-based data types (integers, floating-point values, etc.) are serialized beginning with the high-order byte (the “big” byte).
Byte	A unit of digital information consisting of eight bits.
Compact Reference	A compact version of the standard reference position (see below) intended to reduce memory use by applications accessing a GVRS file. Compact references are stored as four-byte unsigned integers. The value of a compact reference is computed by dividing a reference file position by eight.
CRC32C	A mathematic function for detecting bit errors in data sets. The Cyclic Redundancy Code 32-bit proposed by Castagnoli (1993) is widely used in software implementations and directly support by some hardware architectures.
Descriptive String	A string of characters specified in using the UTF-8 encoding. Descriptive strings are intended for human-readable labels and other text.
Gridfour	The open-source software project for which the GVRS file format was created. The name Gridfour is a portmanteau of the word “grid” and the French word “four” (oven)
Identifier	A string of characters specified using a limited character set (a subset of the ASCII encoding). Identifiers conform to the syntax used for

	variable names in many computer programming and scripting languages.
IEEE-754	A standard for representing floating-point values in binary form.
Label	In GVRS, a label is an arbitrary string intended to identify an element or file component in a human-readable manner. In general practice, labels tend to be short strings that may be used as table headers, in report text, or as user interface components.
Little Endian	A scheme in which byte-based data types (integers, floating-point values, etc.) are serialized beginning with the low-order byte (the "little" byte).
Record	A group of related data elements treated as a coherent unit. In some traditional settings, records were assumed to be of a fixed length (so that all records in a collection were the same size). In GVRS, records are treated as having a variable length.
Reference	A value indicating the position of some element in the file. File positions are specified in byte offsets from the start of the file. The first byte in a file is treated as having a position value of zero. References are stored in the form of eight-byte unsigned integers.
Serialization	The process of converting a larger data element into a sequence of bytes. All serializations are required to be repeatable and reversible.
Timestamp	A representation of time and date measured in milliseconds from the epoch January 1, 1970.
UTF-8 (Unicode)	A variable-width character encoding that covers most modern written languages.

3 Overview

3.1 The data model for a GVRS file

The underlying data model for a GVRS file is the grid. Each cell in a GVRS raster contains one or more data elements. In other words, the cells in a GVRS raster can be treated as *tuples*. These tuples may include any combination of primitive data types (two-byte and four-byte integers, floating-point values, and integer-coded floating-point values). However, the same data definition is used for every cell in the raster.

3.1.1 The virtual grid

The dimensions of gridded data sets can be quite large, often much larger than can be conveniently maintained in memory. To handle very large raster products, GVRS implements a strategy for dividing the overall grid into smaller pieces that are referred to as *tiles.* The tiling concept is illustrated in Figure 1 below. The design of the GVRS file format permits applications to swap tiles in and out of memory as needed. From the perspective of application code, it appears that the entire grid is available in a seamless fashion. A data set managed in this manner is referred to as a *virtual raster.*

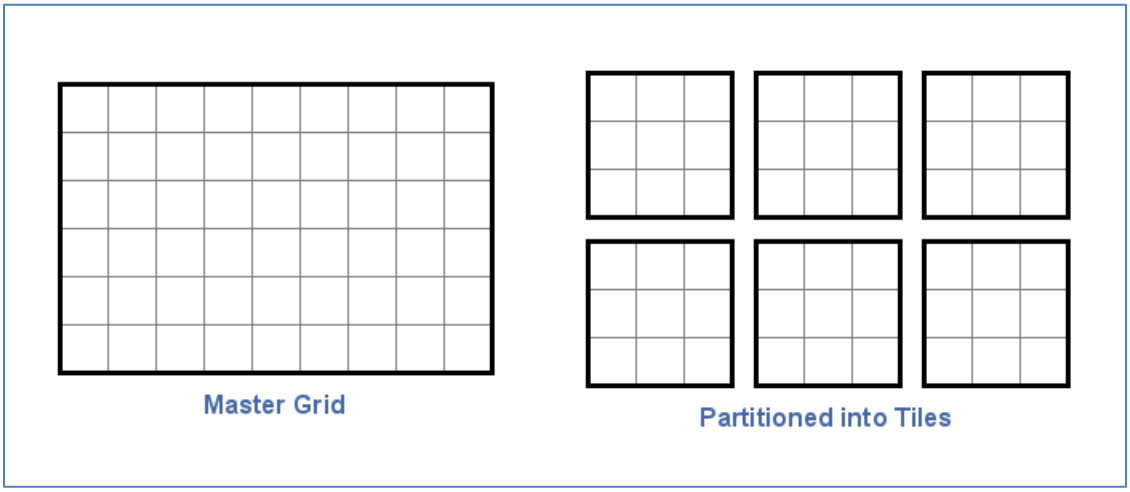


Figure 1 – A tiling scheme.

3.1.2 Grid and tile indexing schemes

The GVRS format treats grids as being indexed in row-major order. Rows, columns, and grid cells are counted started at zero. Thus the first cell in a grid is assigned the index of zero. For a grid of n rows and m columns, the first cell is row 1 is assigned the index of m.

The GVRS format treats tiles as being organized in row-major order. The rows and columns of tiles are counted starting at zero. All tiles are of a fixed, uniform size. In cases where the tile size is not an integral divisor of the overall grid size, the last row and column of tiles may be allowed to extend beyond the grid. Typically, tile cells that extend beyond the bounds of the grid are assigned null values

3.2 The organization of a GVRS file

Figure 2 illustrates the overall organization of a GVRS file. The file begins with a short identification block indicating that it is a GVRS product and indicating which version of the GVRS file format was used to create it. The identification block is followed by a header record that provides information about the file structure and data content. While the header elements are located at a fixed position as the start of the file, the position and size of all other components is arbitrary. The file header maintains references to three directory records which, in turn, maintain references for free space, metadata, and tile records.

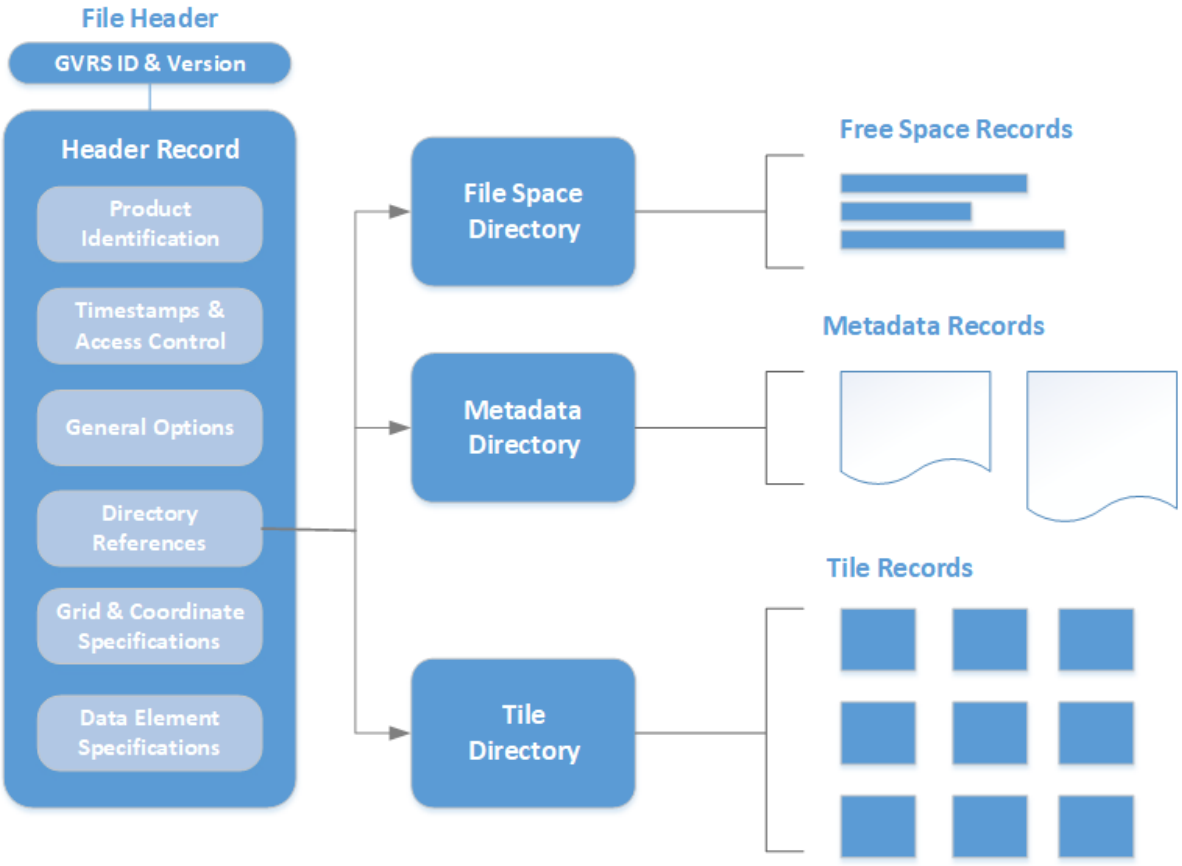


Figure 2 – Main components of a GVRS file.

3.3 The structure of a GVRS record

With the exception of the GVRS ID & Version block, all components in a GVRS file are stored in the form of variable-length records. The structure of a GVRS record is shown in Figure 3 below. A GVRS record begins with an 8 byte *record header* that indicates its length and type. This header is followed by the record content, and finally a checksum.

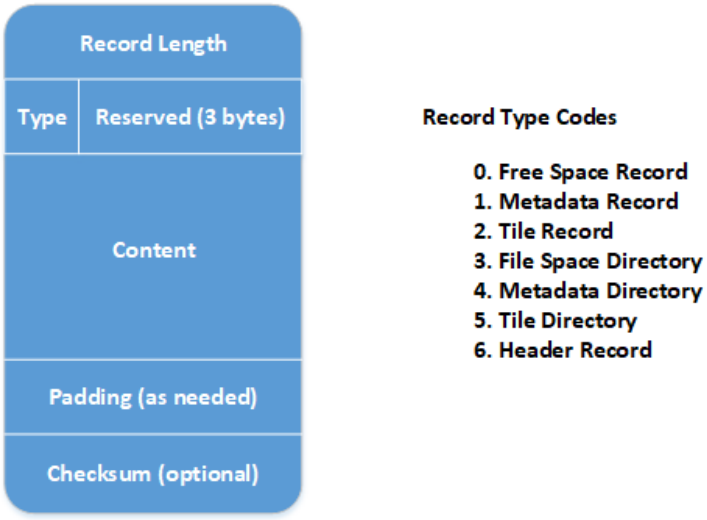


Figure 3 – The structure of a variable-length record.

3.3.1 Record length and padding

Because the records are of variable length, they always begin with an element (a four-byte integer) indicating their size in bytes. Each record is followed by a one-byte code indicating the record type (header, file space directory, metadata directory, tile directory, etc.).

The GVRS format requires that the size of all records be a multiple of eight bytes. In cases where the content of a record does not meet this requirement, padding may be inserted near its end. One consequence of this size requirement is that all GVRS records begin at a file position that is a multiple of eight. The advantage of this approach is discussed below.

3.3.2 Record type and content

A brief description of each record type and its associated content is given in the table below. Detailed information about the layout of each record type is given in section 4.

Table 1 – Record types defined by the GVRS format.

Type	Name	Description
0	Free space record	Indicates an unused block of file space that is available for use.
1	Metadata record	Stores arbitrary data elements attached to a GVRS file by other applications.
2	Tile record	Represents a tile, the primary unit of storage in a GVRS file.
3	File space directory	Provides a directory for tracking free-space. May be omitted if a file does not contain free space records.
4	Metadata directory	Provides a directory of metadata records indexed by metadata name and identification number.
5	Tile directory	Provides a directory of tiles indexed by tile number.

6	Header record	Stores grid specifications, data definitions, product identification, and references to directory records.

3.3.3 The checksum

The last four bytes in a GVRS record contain a checksum value. Because the computation of checksums adds overhead to a file-writing process, their use is optional. A Boolean element in the GVRS header indicates whether a file populates checksums. When checksums are not populated, GVRS requires that they be populated with zeroes.

GVRS uses an industry standard CRC32C checksum. For most record types, the checksum is computed using all bytes in the record except the checksum itself. If the record is valid, the computed checksum will match the value stored at the end of the record. The free-space record type implements a special behavior in which the checksum is computed for only the record header (the first eight bytes of the record). Because the content of a free-space record is undefined, it is not included in a checksum computation.

CRC checksums reliably detect single-bit errors in a data set and have a limited capacity to larger errors. Over the last few decades, the use of embedded checksums in data products has become less important than was formerly the case. Modern file systems and data-transmission protocols implement systems such as Reed-Solomon codes that provide robust error detection and recovery. However, applications that are creating data products for archival storage or distribution may still elect to populate GVRS checksums as an added layer of protection.

3.3.4 File references and record content

In most cases, references in a GVRS file indicate the file position of a record's content, not its header. For example, a reference in the tile directory would point to the content for the corresponding tile. The beginning of the tile record, the position of its header, would occur eight bytes before the content file address.

The exception to this convention is the free-space record. References to free-space records always point to the beginning of the record header.

3.4 Content of the GVRS header record

The first 12 bytes of a GVRS file consist of ASCII characters giving the sequence "gvrs raster" followed by a zero byte (null terminator). This identification is followed by bytes giving the version of the GVRS file. The current version, 1.4, is given by bytes with the value 1 and 4.

This data-format identification is followed by a GVRS header record. The size of this record can vary depending on the complexity of the data elements specified for the file. The content of this record is described below.

3.4.1 Product identification

The first content element in the header record is a set of 16 bytes giving a Universally Unique Identifier (UUID). This element allows an application to attach a unique ID to each GVRS data file. Among other uses, a UUID is suitable for tracking products in digital inventory control systems or establishing a product-creation record. Because a standard API to create a UUID is not available in all development environments, the content of the UUID is optional. The reference implementation uses the Leach-Salz time-based variant (Leach 2005, variant 2). Where feasible, developers are encouraged to conform to this approach.

Toward the end of its header record, a GIVRS file may also include an optional "Product Label". In GVRS, a label is an arbitrary string of UTF-8 characters that is intended for use in human-readable program outputs. The product label supplements the UUID by allowing data authors to attach a plain language identification element to their products. While product labels are optional, it is good practice to include them when a GVRS file is used for persistent storage of data.

3.4.2 Timestamp and access control

The GVRS header includes two timestamps. The first indicates the time that the file was last modified. It is considered a mandatory element and should always have a non-zero value.

The second timestamp is the "opened-for-writing" value. It is populated with the current time when a GVRS file is opened with write-access and it is set to zero when a writeable file is closed. The main purpose of the timestamp is to serve as an access-control mechanism. It allows other applications to detect if a GVRS file is being modified and avoid trying to access it. While a GVRS file is opened for writing, it is in an incomplete state. For purposes of efficiency, most applications will retain critical information in memory and not write it to a file until the file is closed. So, if a file is in an opened-for-writing state, it cannot be shared between applications. Also, if the authoring application terminates without closing the file properly, there is a high probability that the file will be in a damaged state and not suitable for future access.

The opened-for-writing field provides a way of verifying that files are in a proper state when they are provided for data access. Although a single byte Boolean field would suffice for this purpose, GVRS files record a fully qualified timestamp as an aid for troubleshooters who seek to track down the events that led to an improper termination.

3.4.3 General options

The general options indicate the overall behavior and optional features of a GVRS file. For example, the inclusion of computed checksums in a GVRS file is optional. So one of the specifications in the header is whether checksums are enabled. Similarly, when a GVRS file is used for long-term storage, data compression may be preferred as a way to reduce file size. The general options elements in the header indicate whether data compression is used.

3.4.4 Directory references

The file header includes three references to directory components: the file-space directory, the metadata directory, and the tile directory. These references are given as long integers indicating the absolute file position of each directory record. The organization of the file, and the position of its directory records, can change as its content is modified. So the file positions indicated by the directory references may be updated as required. The structure of these directories is discussed in detail later in this document.

3.4.5 Grid and coordinate specifications

GVRS treats its underlying raster as a grid of data cells organized in row-major order. Rows and columns are numbered starting at zero. Cells are indexed using integer values with the first cell being assigned an index of zero.

For many applications, it is useful to be able to overlay a grid with a real-valued coordinate system. The GVRS file header includes elements specifying coordinate transforms (mappings) from real-valued coordinate to grid row and column coordinates.

Details of coordinate systems are discussed later in this document.

3.4.6 Data element specifications

Each cell in a GVRS raster contains one or more data elements. In other words, the cells in a GVRS raster can be treated “tuples”. These tuples may include any combination of primitive data types (two-byte and four-byte integers, floating-point values, and integer-coded floating-point values). However, the same data definition is used for every cell in the raster.

The GVRS file header includes the following information about each element specified for the raster:

- Data Type
- Limits (minimum and maximum values)
- Fill value (the default value for non-populated cells)
- Element name (mandatory GVRS identifier)
- Element label (optional)
- Element description (optional)
- Element unit of measure (optional)

Details of GVRS data element specifications are discussed later in this document.

3.5 Data elements for grid cells

Each cell in a GVRS raster is defined as a tuple containing one or more data elements. GVRS elements are represented using conventional data primitives and are always stored in little-endian byte order. A tuple may contain a mix of data types, but all tuples in the grid are stored using a consistent definition. The data types currently defined for use in raster data cells are given in the table below.

Table 2 – Data types defined for GVRS grid cells.

Type	Size (bytes)	Description
Float	4	An IEEE-754 standard floating-point value (e.g. a single-precision float).
Integer	4	A signed integer value, with negative values given in two's complement form. Range of values: -2147483648 to 2147483647
Short	2	A signed short integer value, with negative values given in two's complement form. Range of values: -32768 to 32767.
Integer-Coded Float	4	Floating point values encoded as integers using a floating point scale and offset value. Floating point values are presented to an application using the GVRS API, but are converted to integers for storage. The integer-coded float format is provided to reflect conventions commonly used in a number of data products found on the Internet. The format is also useful when storing data in a compressed format, because integers usually compress much more effectively than floating-point values.

The following data types are under consideration for future versions of the GVRS specification, but are not implemented at this time.

Table 3 – Data types being considered for future releases.

Type	Size (bytes)	Description
Unsigned short	2	An unsigned short integer value. Values in this format may range from 0 to 65535.
Short-Coded Float	2	Floating point values encoded as unsigned short integers using a floating point scale and offset value. Floating point values are presented to an application using the GVRS API, but are converted to short integers for storage. The short-coded float format is provided to reflect conventions commonly used in a number of data products found on the Internet. The format is also useful when storing data in a compressed format, because integers usually compress much more effectively than floating-point values.
Pixel	4	Planned for future investigation

3.5.1 Element minimum, maximum, and fill values

The GVRS element specification provides minimum, maximum, and fill values. Most API implementations provide default values for minimum and maximum values based on the intrinsic range of the data format. But data authors are strongly encouraged to provide ranges of values that reflect the behavior of their data. Having a meaningful range of values is useful for rendering applications that

assign color palettes to data, modeling applications that perform statistical analysis on data, and for many other practical applications.

The *fill value* is a value that is assigned to all elements in the raster in all cells that have not been explicitly populated by an application or program. The fill value may be interpreted as either a default value or as a null-data value depending on the conventions adopted by the author of the data product.

For example, consider a data product that gives terrestrial elevation and ocean depth values in meters. A feasible range of values for such a data set might be specified as -11000 to 8700. If a developer knew that the product did not cover the entire Earth, he might choose a fill value of -32768 to indicate a no-data value.

On the other hand, consider the example of a product that provided terrestrial elevation data and treated all data cells lying over ocean areas as having an elevation of zero. In that case, a feasible range of values might be specified as 0 to 8700 meters. If the product covered the entire Earth without null values, then the fill value might be specified as zero. By setting the fill value to zero, the product could conserve storage space since areas of ocean data would not have to be recorded in the output file.

3.5.2 Element names

Each element in the tuple definition is associated with a mandatory and unique "name" attribute given as a GVRS identifier string. This name specification allows software implementations to present access methods to application code for obtaining data from a particular element within a particular grid cell.

3.5.3 Element labels and descriptions

The GVRS format allows a data author to specify optional labels and description attributes for individual elements. These attributes are intended to be displayed in user interfaces or printed in automatically generated report documents. Labels are usually short strings indicating the content of an element. Descriptions are usually one or two sentence text elements offering explanatory information about the element. Labels and descriptions are specified using the UTF-8 encoding scheme.

As an example of how the name, label, and description attributes could be used in an application, consider the case of a data set giving elevation information over parts of Brazil. The name of the main element might be specified as either "elevation" or, simply, as "z". To support Portuguese-speaking users, the elevation element might be assigned the label "Elevação" and the description "Elevação em metros sobre o nível médio do ma".

3.5.4 Element unit of measure

The GVRS format allows a data author to specify an optional unit-of-measure string. Unit of measure are expected to conform to widely accepted systems of measure including either SI units, Imperial units, or the United States Customary System. At this time, a rigorous specification for syntax to be used for unit of measure in GVRS has not been determined. Such a specification may be introduced in the future. Therefore, data authors are encouraged to be careful in their selection of unit specifications. On the other hand, when information about unit of measure is available, data authors are strongly encouraged

to include the unit specification as part of their element definitions. Providing unit of measure is essential to ensuring that future uses of data will interpret it correctly.

3.6 Strings

The representation of strings in the GVRS file format uses a compound data type defined as follows:

1. Length (unsigned short): the number of bytes required to encode the string.
2. Content (bytes of specified length): the collection of bytes encoding a string in the UTF-8 format.

GVRS attempts to avoid a European-centric specification for textual information by using the UTF-8 format to specify strings. For most characters in Western alphabets, this does not present a problem since the UTF-8 scheme uses the same byte coding as both the ASCII (American) alphabet and the ISO8859-1 (extended ASCII) character sets. Both of these character sets use exactly one byte per character, so the length specification for a string directly corresponds to the number of characters in the string. Non-European character sets frequently use multiple bytes per individual characters. In such cases, the number of characters in a string will be less than the number of bytes indicated by the length specification.

3.6.1 Identifiers and Descriptive Strings

The GVRS format uses two kinds of strings:

1. Descriptive Strings: Those intended for use by humans.
2. Identifiers: Those intended for use by computer programs.

Descriptive strings are free-form strings that may be used as labels or descriptions. They are intended to be displayed in user interfaces or printed in reports. Descriptive strings are stored using the UTF-8 character encoding. Thus they can represent text using non-Western alphabets.

Identifiers are strings that are used as keys for accessing data elements including numeric values and metadata records. Because they are used by application code, they need to be compatible with the conventions used in scripting languages, relational database tables, and other software systems. Therefore, identifiers are specified using a limited syntax.

The rules for identifiers are as follows:

1. Identifiers use the ASCII character set. No non-ASCII characters are permitted.
2. Identifiers must always be at least one character in length.
3. All identifiers have a maximum length that is defined as part of the GVRS file format. Different features may support different maximum lengths, but all have a maximum length that is less than or equal to 256 characters.
4. Identifiers may contain a combination of ASCII upper and lower cases letters, numbers, or underscores, but identifiers must always begin with a letter.
5. Identifiers are case sensitive.

The syntax for GVRS identifiers resembles that which is used for variables and attributes in various programming languages. Certain additional restrictions are applied to foster portability across software environments. For example, many programming languages permit identifiers to begin with underscores, but GVRS does not. This restriction was adopted to support Python. Python assigns special meaning to variables beginning and ending with underscores. So, GVRS identifiers avoid the use of leading and trailing underscores to avoid potential conflicts with Python implementations.

In the reference Java implementation, methods that accept strings to be used as identifiers apply syntax and length checks to ensure that the specified strings are valid entries. When an input string does not match the restrictions applied by the GVRS format, the methods throw an illegal-argument exception.

4 File format

This section provides details of the GVRS file format in the form of a series of tables describing the content. It begins with a brief discussion of the data primitives used by GVRS and provides specifications for the low-level compound data types such as strings. It then provides details of the component record types used in the GVRS file.

4.1 Data primitives and byte conventions

4.1.1 Bytes and byte order

The fundamental unit of storage for a GVRS file is the byte. All other data types are assumed to be composed of bytes. When bytes are accessed as individual data elements, they are assumed to be unsigned types. When larger data types (integers, floats, etc.) are stored to a GVRS file, they are said to be serialized in little-endian order. When storing a larger data type, GVR writes the low-order byte to the file first. In other words, it starts from the "little end" of the data element. The low-order by is followed by the next lowest-order byte, and so forth. The high-order byte is written last.

4.1.2 Indices for bits and bytes

Occasionally, this specification will need to address the use of specific bits or bytes within the context of a larger data element. In general, the GVRS design tries to avoid the complexity associated with such a fine-grain specifications. But bit-level specifications are sometimes necessary, particularly in data compression applications.

The bits and bytes that make up a data primitive are numbered according to the following rules:

1. Bits are numbered according to their power of 2. So the low-order bit in a 32-bit integer or floating-point value is bit 0. The high order bit is bit 31.
2. Bytes are numbered starting with 0 for the low-order byte and 3 for the high-order byte.
3. The numbering schemes have nothing to do with byte ordering (little-endian versus big-endian orders). They refer to the numeric positions of the bits or bytes within the value being discussed.

4.1.3 Padding for eight-byte alignment and checksum position

One requirement for the design of the GVRS format is that all records in a file must begin on a file position that is a multiple of eight. To meet this requirement, it is necessary that all records in a GVRS file be stored with a size that is a multiple of eight. However the definition for most of the GVRS record types does not necessarily result in a record of this size. Therefore, the GVRS file format introduces a small amount of padding at the end of each record on an as-necessary basis to ensure that their size meets the size requirement.

GVRS also has a requirement that the checksum field in one of its component data blocks (e.g. header or records) be the last thing in that block. Therefore, when padding is necessary, it is introduced before the checksum.

4.1.4 Data primitives used by GVRS

The table below lists the data primitives used by the GVRS file format. All numeric data types used in the GVRS file format are stored in little-endian order.

Table 4 – Data primitives used by GVRS

Type	Size (bytes)	Description
Byte	1	An unsigned integer type consisting of eight bits. Range of values: 0 to 255. Because the Java programming language does not support an unsigned byte type this convention sometimes leads to extra code when manipulating bytes.
Boolean	1	A value of one or zero indicating true (1) or false (0).
Short	2	A signed 16-bit integer type, with negative values given in two's complement form. Range of values: -32768 to 32767.
Unsigned Short	2	An unsigned 16-bit integer type. Range of values: 0 to 65535.
Integer	4	A signed 32-bit integer type, with negative values given in two's complement form. Range of values: -2147483648 to 2147483647.
Unsigned Integer	4	An unsigned 32-bit integer type. Range of values 0 to 4294967296.
Long	8	A signed 64-bit integer type, with negative values given in two's complement form.
Float	4	An IEEE-754 standard 32-bit floating-point type (e.g. a single-precision floating-point type).
Double	8	An IEEE-754 standard 64-bit floating-point type (e.g. a double-precision floating-point type).

4.1.5 String types

The two GVRS string types were introduced earlier in the overview and concepts section of this document. While they differ in terms of syntax, both are stored in the GVRS file using the same structure. The general layout for a string is described below.

Table 5 – Layout of string types used by GVRS

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Length	Unsigned short	0 to 65535	Number of bytes to follow. A zero length is supported.
2	String Content	Byte(Length)	*	A sequence of bytes of the specified length giving the content for the string. Identifier strings are given as a sequence of ASCII characters. Descriptive strings are given as valid UTF-8 character sequences.

4.2 The GVRS file header

4.2.1 The GVRS ID and version block

A GVRS file begins with a header block identifying the file type and version of the specification.

Table 6 – Layout of the GVRS ID and version block

Absolute Position	Field Name	Data Type	Content	Description
0	File Type	ASCII(12)	"gvrs raster"	12-byte string giving GVRS identifier.
12	Version	Byte(1)	1	GVRS file version indicator.
13	Sub-Version	Byte(1)	4	Sub-version identifier.
14	Reserved	Byte(2)	*	Reserved for future use.

4.2.2 The header record

A header record is always stored immediately after the header block. The header record is the only record that is stored at a fixed position. In the table below, file positions are given as offsets measured in bytes from the beginning of the record. For clarity, the table also includes the absolute positions of the corresponding data fields.

Table 7 – Layout of the header record

Offset	Field Name	Data Type	Content	Description
0 (16)	Record Length	Integer	>0	Length of the record, in bytes.
4 (20)	Record Type	Byte(1)	6	Record type indicator.
5 (21)	Reserved	Byte(3)	0	Reserved for future use.
8 (24)	UUID	16 bytes (128 bits)	Application assigned UUID	An arbitrary identification value uniquely assigned to each GVRS file instance. The reference implementation uses the Leach-Salz time-based (variant 2) UUID.
24 (40)	Time Modified	Long	Milliseconds since epoch 1970	Time file was last modified.
32 (48)	Time Opened for Writing	Long	Milliseconds since epoch 1970	Indicates the time that the file was last opened for writing. Required to be set to zero when the file is closed. An open file should not be accessed by other applications. A non-zero value may indicate an improper termination of a data-writing application.

40 (56)	Offset to File-Space Directory	Long	File position in bytes	Indicates the file position for the *content* of the file-space directory record. A value of zero indicates that no unallocated file space exists.
48 (64)	Offset to Metadata Directory	Long	File position in bytes	Indicates the file position for the *content* of the metadata directory. A value of zero indicates that no metadata exists.
56 (72)	Number of Levels	Short	0	Number of levels in file, not currently used. Reserved for representing pyramid structures in future GVRS specifications.
58 (74)	Reserved	Byte(6)	0	Reserved for future use.
64 (80)	Offset to Tile Directory	Long	File position in bytes	Indicates the file positon for the *content* of the tile directory.
72 (88)	Reserved	Byte(16)	0	Reserved for future use.
88 (104)	N Rows in Raster	Integer	>=1	Number of rows in the raster grid.
92 (108)	N Columns in Raster	Integer	>=1	Number of columns in the raster grid.
96 (112)	N Rows in Tile	Integer	>=1	Number of rows in a tile.
100 (116)	N Columns in Tile	Integer	>=1	Number of rows in the overall raster grid.
104 (120)	Reserved	Byte(8)	0	Reserved for future use.
112 (128)	Checksum Enabled	Boolean	T/F	Indicates that checksums are populated with meaningful values (checksums should be populated with zero values if this flag is false).
113 (129)	Raster Space Type	Byte(1)	0:2	Indicates how values of raster cells should be interpreted. 0. Unspecified 1. value is point 2. value is area
114 (130)	Coordinate System Type	Byte(1)	0:2	Indicates what kind of real-valued coordinate system is specified for the raster grid. 0. Unspecified 1. Cartesian 2. Geographic
115 (131)	Reserved	Byte(5)	0	Reserved for future use.

120 (136)	X min (x0)	Double	*	Real-valued coordinate associated with the domain of the raster grid (zero if unspecified)
128 (152)	Y min (y0)	Double	*	Real-valued coordinate associated with the domain of the raster grid (zero if unspecified).
136 (160)	X max (x1)	Double)	*	Real-valued coordinate associated with the domain of the raster grid (zero if unspecified).
144 (168)	Y max (y1)	Double	*	Real-valued coordinate associated with the domain of the raster grid (zero if unspecified).
152 (176)	Cell Size X	Double	*	Width of the cell (distance across grid columns)
160 (176)	Cell Size Y	Double	*	Height of the cell (distance across grid rows).
168 (184)	M2R00	Double	*	Affine Transform element for Model-to-Row transformation, row 0, column 0.
176 (192)	M2R01	Double	*	Affine Transform element for Model-to-Row transformation, row 0, column 1.
184 (200)	M2R02	Double	*	Affine Transform element for Model-to-Row transformation, row 0, column 2.
192 (208)	M2R10	Double	*	Affine Transform element for Model-to-Row transformation, row 1, column 0.
200 (216)	M2R11	Double	*	Affine Transform element for Model-to-Row transformation, row 1, column 1.
208 (224)	M2R12	Double	*	Affine Transform element for Model-to-Row transformation, row 1, column 2.
216 (232)	R2M00	Double	*	Affine Transform element for Row-to-Model transformation, row 0, column 0.
224 (240)	R2M01	Double	*	Affine Transform element for Row-to-Model transformation, row 0, column 1.
232 (248)	R2M02	Double	*	Affine Transform element for Row-to-Model transformation, row 0, column 2.
240 (256)	R2M10	Double	*	Affine Transform element for Row-to-Model transformation, row 1, column 0.

248 (264)	R2M11	Double	*	Affine Transform element for Row-to-Model transformation, row 1, column 1.
256 (272)	R2M12	Double	*	Affine Transform element for Row-to-Model transformation, row 1, column 2.
264 (280)	Number of Tile Date Element Specifications	Integer	>=1	Number of tile element specifications.
268 (284)	Tile Element Specifications	Complex	*	Specifications for tile data element(s). See Table 8.
*	Number of Data Compression Methods	Integer	>=0	Number of data compression methods used for file (zero if data compression is not applied).
*	Compression Method(s)	Identifier(s)	Names of data compression methods	If compression methods are defined, a series of GVRS Identifier strings indicating data compression method. The maximum name length for a codec identifier is 32 ASCII charaters.
*	Product Label	UTF-8 String	Variable length	A free-form string intended to serve as a label for the product in user interfaces, reports, and other media.
*	Reserved	Byte(8)	0	Reserved for future use.
*	Padding	Byte(*)	0	Zero to 7 bytes as required to ensure that the record size is a multiple of eight.
*	Checksum	Integer	*	If checksums are used, a CRC32C checksum computed for all bytes in the record, including the header, that precede the checksum. If checksums are not used, a zero.

4.2.2.1 The Element specification type

Each cell in the GVRS raster grid is treated as a tuple containing one or more data elements. The following table describes the structure of an element specification. The offsets listed in the table are added to the file position at the state of the element.

Table 8 – Layout of an element specification entry.

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Element Data Type	Byte(1)	*	The data type for the element: 0. Integer 1. Integer-coded float 2. Float

				3. Short
1	Continuous	Boolean	T/F	Indicates whether the element is interpreted as a member of a continuous field (true) or a discrete-valued function (false).
2	Reserved	Byte(6)	0	Reserved for future use.
8	Name	Identifier	*	A mandatory, non-empty name for the element. Maximum length 32.
*	Element range and fill value specifications	Complex	*	A type-specific specification for minimum, maximum, fill value, and other values as required. See Table 9, Table 10, Table 11, and Table 12
*	Fill	Byte(*)	0	Zero to 3 bytes as required to ensure that the file position is set to a multiple of 4.
*	Label	UTF-8 String	*	If provided, a descriptive string providing a human-readable label for the element. If omitted, a zero-length string.
*	Description	UTF-8 String	*	If provided, a descriptive string providing human-readable information about element. If omitted, a zero-length string.
*	Unit of Measure	ASCII String	*	If provided, an ASCII-formatted string giving the unit of measure. If omitted, a zero-length string.
*	Fill	Byte(*)	0	Zero to 3 bytes as required to ensure that the specification size is a multiple of 4 (the element specification is byte-aligned for four-byte data types).

4.2.2.2 Element value specification for integer data types

If the element has a data type of Integer, the following specifications are provided within the Tile Element specification block.

Table 9 – Layout of the element range and fill values specification (Integer).

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Min Value	Integer	-2147483648 to 2147483647	The minimum value allowed for the element.
4	Max Value	Integer	-2147483648 to 2147483647	The maximum value allowed for the element
8	Fill Value	Integer	-2147483648 to 2147483647	The fill value for the element

4.2.2.3 Element value specification for float data types

If the element has a data type of Float, the following specifications are provided within the Tile Element specification block.

Table 10 – Layout of the element range and fill values specification (Float).

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Min Value	Float	Negative Infinity to Positive Infinity	The minimum value allowed for the element.
4	Max Value	Float	Negative Infinity to Positive Infinity	The maximum value allowed for the element
8	Fill Value	Float	Negative Infinity to Positive Infinity or Float.NaN	The fill value for the element

4.2.2.4 Element value specification for integer-coded- float data types

If the element has a data type of Integer-coded float, the following specifications are provided within the Tile Element specification block.

Table 11 – Layout of the element range and fill value specification (Integer-coded float).

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Min Value	Float	Range of finite floating point values	The minimum value allowed for the element.
4	Max Value	Float	Range of finite floating point values	The maximum value allowed for the element
8	Fill Value	Float	Range of finite floating point values or Float.NaN	The fill value for the element
12	Scale	Float	Range of finite floating point values	The scaling value converting a floating point value to an integer, and vice versa.
16	Offset	Float	Range of finite	The offset value for converting a

			floating point values	floating point value to an integer, and vice versa
20	Min Value	Integer	-2147483648 to 2147483647	The minimum integer value for the element.
24	Max Value	Integer	-2147483648 to 2147483647	The maximum value allowed for the element
28	Fill Value	Integer	-2147483648 to 2147483647	The fill value for the element

4.2.2.5 Element value specification for short data types

If the element has a data type of Short, the following specifications are provided within the Tile Element specification block.

Table 12 – Layout of the element range and fill value specification (Short).

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Min Value	Short	-32768 to 32767	The minimum value allowed for the element.
4	Max Value	Short	-32768 tp 32767	The maximum value allowed for the element
8	Fill Value	Short	-32768 to 32767	The fill value for the element

4.2.3 The free-space record

If content is removed from a GVRS file, its former position is marked as a “free-space record”. The organization of a free space record is as follows.

Table 13 – Layout of a free space record.

Offset	Field Name	Data Type	Content	Description
0	Record length	Integer	>0	Length of the record, in bytes
4	Record type	Byte	0	Indicates that the record is a free-space record.
5	Reserved	Byte(3)	0	Reserved for future use
8	Empty Content	Byte(*)	*	Undefined. For information assurance and security purposes, developers are strongly encouraged to store zeroes in these locations.
*	Checksum	Integer	*	If checksums are used, a CRC32C checksum computed for all bytes in the record the header. If checksums are not used, zero.

The checksum calculation for the free-space uses only the record header (the first eight bytes). This approach is different than that used for other record types which consider the entire content of the record. Because the content of the free-space record is undefined, it is not considered in the checksum.

4.2.4 The metadata record

A metadata entity is defined by a unique compound key consisting of an identifier and a record ID. The maximum length for the identifier is 32 characters. Note that while multiple metadata entities may have the same name, the combination of name and identifier must be unique.

Table 14 – Layout of a metadata record.

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Record length	Integer	>0	Length of the record, in bytes
4	Record type	Byte	1	Indicates that the record is a metadata record
5	Reserved	Byte(3)	0	Reserved for future use
8	Name	Identifier	*	A non-empty string identifying the metadata entity, maximum length 32 characters.
*	Record ID	Integer	*	An integer identifying the metadata entity.
*	Data Type	Byte	*	The data type for the information stored within the metadata content. 0. Unspecified 1. Byte 2. Short 3. Unsigned short 4. Integer 5. Unsigned integer 6. Float 7. Double 8. String (UTF-8) 9. ASCII
*	Content Length	Integer	>=0	If content is provided, length of content, in bytes. The length is always in bytes irrespective of the datatype for the metadata.
*	Content	Byte (Content Length)	*	If content is provided, a sequence of bytes containing content.
*	Description	UTF-8	*	If provided, a descriptive string providing human-readable information about element. If omitted, a zero-length string.
*	Padding	Byte(*)	0	0 to 7 bytes as required to make the overall size of the tile a multiple of

				eight.
*	Checksum	Integer	*	If checksums are used, a CRC32C checksum computed for all bytes in the record, including the header, that precede the checksum. If checksums are not used, zero.

4.2.5 The tile record

All tile records begin with a standard header followed by an integer giving the tile index for the tile. Although the tile index could be deduced from other data elements present in the application, it is treated as a mandatory field and must be populated with the correct value. This practice permits the tile index to be used for diagnostic and error-recovery operations.

Table 15 – Layout of a tile record.

Offset	Field Name	Data Type	Content	Description
0	Record length	Integer	>0	Length of the record, in bytes
4	Record type	Byte	2	Indicates that the record is a tile record.
5	Reserved	Byte(3)	0	Reserved for future use.
8	Tile Index	Integer	>=0	An integer giving the index of the tile.
12	Tile Elements	Complex Type	*	Variable depending on data type, number of elements in specification, and whether data compression is applied (see below)
*	Padding	Byte(*)	0	0 to 7 bytes as required to make the overall size of the tile a multiple of eight.
*	Checksum	Integer	*	If checksums are used, a CRC32C checksum computed for all bytes in the record, including the header, that precede the checksum. If checksums are not used, zero.

4.2.5.1 Storage format for tile elements

Each tile may contain one or more elements. Elements may be stored in either compressed or non-compressed form. The definition of the elements, including the datatype and storage size for uncompressed forms, is provided by the GVRS header record. The definition of elements is the same for all tiles in a GVRS file.

Tile elements are stored in sequence. An “interleaved” format is not used. If a tile includes multiple elements, the first element will be stored in its entirety, followed by the second element, etc. If data compression is applied, the set of values for each element is compressed independently. The general layout for tile elements is shown in the table below.

Table 16 – Layout for the element storage section of a tile.

Offset	Field Name	Data Type	Content	Description
0	Element Storage Length	Integer	*	The length of the data to follow, in bytes
12	Element Content	Complex Type	*	Variable. Depends on data type and whether data compression is applied
	Padding	Byte(*)	0	0 to 3 bytes as required to ensure that the overall size of the element is a multiple of ***four***. This padding should not be confused with the overall padding for the tile record. It is part of the content storage.

The padding for tile elements is based on byte alignment with a multiple of 4 (the size of the length field for the element content). Most tile data types require four bytes per data value. In their non-compressed form, they will not require padding. The short data type uses two bytes per value. So a tile containing an odd number of short values requires two bytes of padding. Compressed data may be of an arbitrary length and will often require padding.

4.2.5.2 Element storage length indicates whether data compression is used

Even when data compression is enabled, it may not always be feasible to store the data for an element in compressed form. For example, an input data set containing a high degree of randomness may not be compressible. In such cases, the element content will be stored in non-compressed form.

It is possible for a file to contain a mix of compressed and non-compressed tiles. It is possible for a tile to contain a mix of compressed and non-compressed element content.

The element length may be used to determine whether data is stored in compressed form. If the element content length is equal to the number of bytes that would be required to store the element content in its non-compressed format, the data must be non-compressed. If the content length is smaller than the number of bytes to store the data in its non-compressed form, then the data must be compressed. The content length is never allowed to be larger than the uncompressed size of the content.

4.2.5.3 Storage format for uncompressed element content

The uncompressed element content is always stored as a sequence of values encoded according to the data type of the element specification. Data is always stored in row-major order and in little-endian byte order.

Integer-Coded Floating-Point values are stored in their scaled integer format.

4.2.5.4 Storage format for compressed element content

The GVRS implementation includes support for a number of different compression formats. These are described in the Appendices below. In some cases, there may be multiple choices of a compression format for a particular set of tile elements. Therefore, a compressed sequence must be introduced by an index value indicating which compression method is used.

Compressed data sequences always start with a one-byte index value that indicates the compression type.

A GVRS file specification may include up to 256 different compression formats, though only a small number of compressors are implemented at this time. The actual selection of compressors can vary according to the type of data stored in a file and the requirements of the application that uses them.

In practice, the overall file specification will include a metadata element indicating which compressors are in effect. The index element at the start of the tile will indicate which of these are applied.

4.2.6 The file-space directory record

The GVRS specification permits data to be written to a file in an arbitrary pattern. It is also possible for an application to open and modify an existing file. The content of a tile or metadata record may change multiple times in the life time of a file. The size of an existing record may change as new data is added or old data is removed. Thus GVRS requires a method to keep track of what file space is used and what space (if any) is available for re-use.

The file-space directory keeps a record of what sections of the file are in use. It a file is opened for write access, the File-Space Directory will be stored when the file is closed. If an existing file is opened for write access, it will be read from the file when the file is opened. The layout of the file-space directory record is given in the table below.

Table 17 – Layout of the file-space directory.

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Record length	Integer	>0	Length of the record, in bytes
4	Record type	Byte	3	Indicates that the record is a file-space directory.
5	Reserved	Byte(3)	0	Reserved for future use
8	Free-space record count	Integer	>0	The number of blocks of free space within the file.
12	Free-space record reference	Complex Type (Free Block Count)	*	One or more complex elements indicating the position and size of a free-space record (see below)
*	Padding	Byte(*)	0	0 to 7 bytes as required to make the overall size of the tile a multiple of eight.

*	Checksum	Integer	*	If checksums are used, a CRC32C checksum computed for all bytes in the record, including the header, that precede the checksum. If checksums are not used, zero.

4.2.6.1 Format of a free-space record reference

The file-space directory contains one or more complex elements giving the position and size of the free space record.

Table 18 – Layout of a free-space reference entry.

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Free Block Position	Long	>0	A file position pointing to the *beginning* of a free-space record. This reference value is different from other GVRS references in that it indicates the beginning of the free-space record, rather than the beginning of its content.
8	Free Block Size	Integer	0 or larger	The size of the free block

4.2.7 The metadata directory record

The layout of the metadata directory record is shown below.

Table 19 – Layout of the metadata directory record.

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Record length	Integer	>0	Length of the record, in bytes.
4	Record type	Byte	4	Indicates that the record is a metadata directory
5	Reserved	Byte(3)	0	Reserved for future use
8	Metadata Record Count	Integer	0 or larger	The number of metadata records stored in the file
12	Metadata Record Reference	Complex Type (Metadata Record Count)	*	One or more complex elements indicating the position, name, and record ID for a metadata element.
*	Padding	Byte(*)	0	0 to 7 bytes as required to make the overall size of the tile a multiple of eight.
*	Checksum	Integer	*	If checksums are used, a CRC32C checksum computed for all bytes in the record, including the header, that

				precede the checksum. If checksums are not used, zero.

4.2.7.1 Format for a metadata record reference

The header information for the record is followed by metadata-record count instances of free metadata-record references. The format for metadata-record references is given below.

Table 20 – Layout of a metadata record reference entry.

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Metadata File Position Reference	Long	>0	A file position
8	Name	Identifer	*	An identifier giving the name of the metadata element, maximum length 32.
*	Record ID	Integer	*	An arbitrary identification value associated with the record
*	Metadata Type Code	Byte	*	The data type for the information stored within the metadata content. 0. Unspecified 1. Byte 2. Short 3. Unsigned short 4. Integer 5. Unsigned integer 6. Float 7. Double 8. String (UTF-8) 9. String (ASCII)

4.2.8 The tile directory record

At this time, the GVRS format only defines one format for tile directory. While this format is suitable for many applications, there may be scenarios that would be better served by a different specification. Therefore, the tile directory record includes a one-byte code indicating which format is used. At this time, the format code is always set to zero.

To expedite data access, it is common for application code to store the entire tile directory in memory. Furthermore, a large grid may require the specification of a large number of tiles. To conserve memory, GVRS implements a *compact reference* for file positions. A compact reference is computed by dividing the position of a tile record by eight. Because the file position at which GVRS records are stored is always an integral multiple of eight, dividing tile record positions by eight results in an integer value with

no loss of information. The original value may be recovered by multiplying the compact reference by eight.

It is common for a GVRS file with a large grid size to only include data values for a sub-region of the main grid. In such cases, it may be advantageous to store tile references for just the populated region. This approach is supported by the row0, col0, Number of Rows, and Number of Columns fields

Table 21 – Layout of the tile directory record.

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Record length	Integer	>0	Length of the record, in bytes.
4	Record type	Byte	5	Indicates that the record is a tile directory.
5	Reserved	Byte(3)	0	Reserved for future use.
8	Tile-Directory Format	Byte	0	The tile-directory format. Currently, this value is always zero.
9	Standard References are Used	Boolean	T/F	Indicates that standard, eight-byte file position references should be used rather than the compact four-byte reference. Because the compact form is preferred, this value is false, except where required by very large file sizes.
9	Reserved	Byte(6)	0	Reserved for future use
16	Row0	Integer	>=0	The grid index for the first row of tiles stored in the directory.
20	Col0	Integer	>=0	The grid index for the first column of tiles stored in the directory.
24	Number of Rows	Integer	>0	The number of rows of tiles represented by the directory.
28	Number of Columns	Integer	>0	The number of columns of tiles represented by the directory.
32	File Position Reference	Unsigned Integer or Long	>=0	Indicates the file position of a tile. A file position of zero indicates that a tile is not populated and not stored in the file. The number of entries in this field depends on the number of rows and columns of tiles specified above.
*	Padding	Byte(*)	0	0 to 7 bytes as required to make the overall size of the tile a multiple of eight.
*	Checksum	Integer	*	If checksums are used, a CRC32C checksum computed for all bytes in the record, including the header, that precede the checksum. If checksums are not used, zero.

5 Data compression

The GVRS software library implements two broad classes of data compression:

1. Data compression that operates over integer data
2. Data compression that operates over floating-point data

In practice, integer data tends to compress more readily than floating-point data. Also, techniques that work well for integer forms are often ineffective for floating-point values. Therefore, GVRS treats these two forms separately.

The information in this section provides an introduction to the concepts implemented in the GVRS software library. Additional discussions of the ideas used in GVRS are available at the Gridfour Documentation page in the following articles:

1. Compression Algorithms for Raster Data used in the Gridfour Implementation at https://gwlucastrig.github.io/GridfourDocs/notes/GridfourDataCompressionAlgorithms.html
2. Lossless Compression for Floating-Point Data at https://gwlucastrig.github.io/GridfourDocs/notes/LosslessCompressionForFloatingPointData.html
3. Lossless Compression for Raster Data using Optimal Predictors at https://gwlucastrig.github.io/GridfourDocs/notes/CompressionUsingOptimalPredictors.html

Details of the storage format for compressed data are given in the sections that follow.

5.1 Data compression for integer data

The operation of a GVRS data compressor for raster products consisting of integers can be divided into three sections

1. A predictive technique is used transform the data into a sequence of integers having small magnitudes and an increased degree of redundancy (e.g. a reduced information entropy).
2. The integers are serialized into bytes using a custom scheme known as M32 codes.
3. The M32 codes are compressed using a conventional data compression scheme such as Huffman coding or the popular Deflate (zip) API.

5.1.1 Predictive techniques

Data compression techniques that work well for text often yield poor results for raster data sets. Virtually all data compression algorithms operate by identifying redundant elements in a data set and replacing them with more compact representations. Unfortunately, many raster data sets, particularly those containing geophysical information, tend to not present redundancy in a form that conventional data compression tools can exploit.

GVRS addresses the problem of non-compliant raster data by using a technique called predictive modeling. GVRS implements models that predict the value at each grid point in the

raster. The residuals from these predictions (actual value minus predicted value), tend to be small in magnitude and often reveal a high degree of redundancy. Thus, they are more readily compressed than the source data. In its compressed form, the data is represented using the prediction parameters and the compressed residuals. During decompression, these residuals are used as correction factors that adjust the predicted values to match the original inputs.

5.1.2 A simple predictor

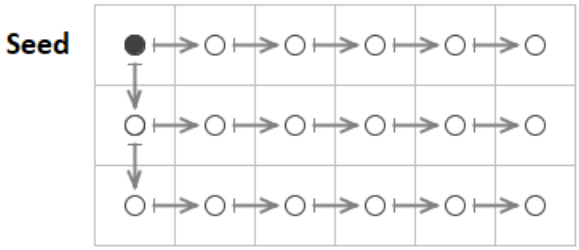
The most elementary predictor for a raster data set is one that simply assumes that the value of each grid cell in a series is simply that of the cell that immediately preceded it. In that case, the computed residuals are simply the differences between neighboring grid cells. In the GVRS implementation, this approach is referred to as the *differencing predictor.*

When we consider the Differencing Predictor Model over a single row of data in a grid, identifying neighbors is straight forward. The relevant neighbor point is just the previous or next data point in the sequence. But special handling is required when processing the transition from the end of one row to the beginning of the next. If taken in sequence, these two sample points will not be spatially correlated.

In practice, this requirement for special handling is easily met. In the uncompressed form, GVRS stores grid points in row-major order (one row at a time). So in most cases, the predecessor of a grid point is just the sample that preceded it in the row. There is, however, one edge case that requires special handling. In row-major order, the grid point that follows the last point in a row is the first point in the next row. So, a predictor-residual based on the Differencing Model needs to implement special handling for that transition. In GVRS, the following rules are applied:

1. The first grid point in the first row of a tile is treated as a "seed" value.
2. The difference value for each point in a row, except the first, is computed using the grid point that preceded it.
3. The difference values for the first grid point in all rows (except the first row), are computed using the first value in the row that preceded it.

The figure below illustrates the pattern. The seed value is shown as a solid dot, the delta values are all shown as circles. The arrows indicate which samples are paired together to compute delta values.



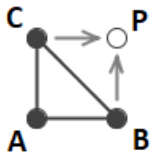
It is worth noting that the use of the Differencing Model for raster data is not unique to thr Gridfour project. It is used in a number of specifications including the GRIB2 raster data format (NCEP 2005, table 5.6) and the TIFF image format (Adobe, 1992, p. 64, "Section 14: Differencing Predictor").

5.1.3 Other predictors

The main GVRS API implements two additional predictors: the Linear Predictor and the Triangle Predictor.

The Linear Predictor model predicts that the data varies as a linear function. The value for the next sample in a sequence is predicted using a straight-line computed from the two that preceded it. The predictor is applied on a row-by-row basis. The vertical coordinate of the points that precede the target point are assigned the values Z_A, Z_B respectively. If we assume that the grid points are spaced at equal intervals, then the predicted value, Z_P, is given by $Z_P = 2xZ_B-Z_A$.

The Triangle Predictor was described by Kidner & Smith (1992). It uses three neighboring points A, B, and C, to predict the value of a target sample as shown in the figure below. The vertical coordinate of the points are assigned the values Z_A, Z_B, and Z_C respectively. By treating the grid as having a fixed spacing between columns, *s*, and a fixed spacing between rows, *t*, the prediction computation is simplified to $Z_P = Z_B+Z_C-Z_A$.



5.1.4 Layout for standard predictors

The following tables illustrate the layout for the standard predictors. The example layouts are formed from a set of 16 integral values labeled according to the English alphabet (A, B, C, …, P)

Source data

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

The differencing predictor

A	B-A	C-B	D-C
E-A	F-E	G-F	H-G
I-E	J-I	K-J	L-K
M-I	N-M	O-N	P-O

The linear predictor

```
A       B-A     C-(2B-A)        D-(2C-A)
E-A     F-E     G-(2F-E)        H-(2G-F(
I-E     J-I     K-(2J-I)        L-(2K-J)
M-I     N-M     O-(2N-M)        P-(2O-N)
```

The triangle predictor

```
A       B-A             C-B             D-C
E-A     F-(B+E-A)       G-(C+F-B)       H-(D+G-C)
I-E     J-(F+I-E)       K-(G+J-F)       L-(H+K-G)
M-I     N-(J+M-I)       O-(K+N-J)       P-(L+O-K)
```

5.1.5 Serialization for residuals

Once the residuals are computed using a predictor-residual model, they are passed to compression processes based on conventional data compression algorithms such as the well-known Huffman coding scheme or the standard Deflate techniques which is used for the ZIP file format and for several other applications. Both the custom Huffman implementation included with the GVRS code base and the Deflate API provided as part of the standard Java library are designed to process bytes. But the predictive-residual models produce output in the form of integers. So in order to use them to process and store the outputs from the models, the residuals must somehow be serialized into a byte form. Most of the residuals tend to be close to zero, so the serialization for them is trivial. In some cases, however, the residuals will be in excess of the value that can be stored in a single byte.

While it would be feasible to simply split out the integer residual values into the component bytes, doing so would tend to dilute the redundancy in the output data.

To serialize the residuals, GVRS uses a scheme that it calls the M32 code. M32 is an integer-to-byte coding scheme that adjusts the number of bytes in the output to reflect the magnitude of each term in the input. In that regard, it uses an approach similar to that used for the widely used UTF-8 character encoding. As in the case of UTF-8, the first byte in the sequence can represent either a literal value or a format-indicator that specifies the number of bytes to follow. M32 also resembles the variable-length integer code used by SQLite4 (SQLite4, 2019), except that it supports negative values as well as positive. The first byte in the output is essentially a hybrid value. For small-magnitude values (in the range -126 to +126), it is just a signed-byte representation of the input value. The values -127 and +127 are used to introduce multi-byte sequences. And the value -128 is used to indicate the minimum 32-bit integer value, -2147483648, which GVRS often uses as a null-data code. In the multi-byte sequences, the bytes that follow the introducer give bits for the numerical value in *big-endian order*. Each byte carries 7 bits of information with the high-order bit used to indicate if additional bytes follow. The length of the sequence depends on the magnitude of the residual to be coded. In the worst case, 6 bytes are required to encode large magnitude residual (1 byte for the introducer, 5 bytes for the content).

The M32 coding scheme is effective in the case where the computed residuals tend to be small because it preserves the one-byte-per-value relationship. Again, in the case of larger residuals, the M32 code can actually be longer than the 4 bytes needed for a simple integer. Fortunately, a good predictor will produce small residuals.

5.2 Data compression for floating-point raster data

Some implementations attempt to compress floating-point data by treating it as a sequence of bytes. One of the drawbacks to this approach is that the bytes in a floating point representation mix different kinds of data. For example, in the IEEE-754 single-precision floating point format (4 bytes), the layout of the representation is

- Byte 3 (high-order byte): sign bit and the seven high-order bits from the exponent
- Byte 2: one bit from the exponent, the seven high-order bits from the mantissa
- Byte 1: eight bits from the mantissa
- Byte 0 (low-order byte): low eight bits from the mantissa

Often, the three components of a floating point value have much different statistical properties. Conflating pieces of the different components into a serialized byte stream obscures the redundancy and predictability of the data and makes it harder to compress. Additionally, floating-point data values often feature a great deal of statistical noise, particularly in the low-order components of the mantissa.

The GVRS library improves the data compression for floating-point values by breaking the component pieces into separate data sets which are compressed separately. In some cases, a differencing predictor is used to improve the redundancy of the data and reduce its compressed size. The component groups are as follows:

1. Sign bits (1 bit per value)
2. Exponent (8 bits per value), no differencing applied
3. Mantissa high-order 7 bits, differencing applied
4. Mantissa middle 8 bits, differencing applied
5. Mantissa low-order 8 bits, differencing applied.

When differencing is applied, the calculate follows the pattern described for the standard differencing predictor (see 5.1.4 Layout for standard predictors).

```
A      B-A    C-B    D-C
E-A    F-E    G-F    H-G
I-E    J-I    K-J    L-K
M-I    N-M    O-N    P-O
```

The choice of whether a component group is processed using differencing is based on experimentation with a number of data sets including elevation/bathymetry data sources and artificial test data. So it is worth emphasizing that the choice is not based on any formal theory of data compression. It was established through plain old trial-and-error. In view of that, the choices made for GVRS may not be the optimal solution for all data sets.

6 Format for compressed data within a tile

As described above, each compressed element in a tile is introduced by a 4 byte integer indicating the number of bytes required to store the data. If the byte count is smaller than the standard uncompressed size of the tile, then it is assumed to be compressed.

All compression sequences begin with a single byte called the *compressor index* that indicates which compressors are used. The standard integer compression sequences (Huffman and Deflate) follow with 9 bytes providing initialization data as indicated in the table below. Custom compressors are free to define the encode sequences following the compressor index according to their own requirements.

Offset (Bytes)	Field Name	Data Type	Content	Description
0	Compressor Index	Unsigned Byte	0 to 255	Indicates which compressor was used to store the data. The index refers to a set of compressor names stored in the GVRS file header. The meaning of the index may vary from file to file (i.e. and index of 1 may be Huffman for one file and Deflate for another).
1	Predictor Index	Unsigned Byte	0 to 255	A value indicating which predictor is used: 0. Simple differencing 1. Linear predictor 2. Triangle predictor
2	Seed	Integer	*	An integer value to be stored in the first cell in the grid. Also used to initialize the prediction sequence
6	nM32 Codes	Integer	>0	The number of distinct M32 codes. Because some M32 codes are multi-byte sequences, this value may be less than the remaining number of bytes in the compression sequence.

The remainder of the bytes in the compression sequence represents compressed bytes in either the Huffman coding form or in the standard Deflate encoding. When decompressed, these bytes will yield a sequence of bytes in the M32 coding format. The Deflate encoding is a standard format which is outside the scope of this document. The GVRS Huffman coding format and the M32 coding format are unique to GVRS and are described below.

6.1 Deflate format

The Deflate format follows the conventions established by the standard Deflate API (Deutsch, 1996). A discussion of the Deflate format is outside the scope of this document.

6.2 Huffman coding format

The Huffman coding algorithm (Huffman, 1952) is well known and is discussed extensively on the web. While some implementations of the Huffman algorithm record a frequency table to permit the construction of an encoding tree, many others elect to store the structure of the tree itself. GVRS uses this later approach.

In GVRS, the Huffman tree is stored according to the following recursive algorithm which starts from the root note of the tree.

1. Introduce the sequence with an unsigned byte indicating the number of unique symbols, N, in the tree. Since there will never be zero symbols in the tree, GVRS stores the value N-1 in this byte. Thus one byte is sufficient to represent all 256 possible symbol sets that could arise in a M32 representation.

2. Traverse the Huffman tree starting from the root node using RecusiveStore(rootNode)

3. Function RecursiveStore(node)
   a. If the node is a branch
      i. Output a bit with value 0.
      ii. Call RecursiveStore on the left child node.
      iii. Call RecursiveStore on the right child node.
      iv. Return.
   b. If the node is a leaf (a terminal)
      i. Output a bit with a value 1.
      ii. Output the symbol.
      iii. Return.

Once the tree is established in memory, compressed sequences can be decoded using simple tree traversal. In an encoded bit sequence, a bit value of zero is interpreted as a left traversal. A bit value of one is interpreted as a right traversal. When a leaf node (terminal node) is reached, the symbol that is stored in the leaf is added to the decoded output.

7 Serializing integer data using M32 codes

Both the Deflate and Huffman data compressors operate on bytes. In order to use them to compress integer data, the integers must first be serialized into a sequence of bytes. The most basic form of serialization is to simply split out the component bytes for each integer and arrange them in a sequence. But, when processing residual data from a predictor, the basic approach has a significant disadvantage. Most of the serialized residuals will be of small magnitude. In testing, we've seen cases where 99 percent of the residuals were within the range -128 to +127 and could be expressed as a single signed byte. Storing these values with more than one byte is wasteful. At the same time, large-magnitude residuals that cannot fit within a single byte do occur and must be accommodated. So an effective compressor requires a serialization scheme that is efficient with small values byte and still allows for large values.

The obvious solution for this requirement is to use an encoding system that adapted to the magnitude of the values it needed to store. GVRS uses a scheme called the "M32" code. M32 is an integer-to-byte serialization scheme that adjusts the number of bytes in the output to reflect the magnitude of each term in the input. In that regard, it uses an approach similar to that used for the widely used UTF-8 character encoding. As in the case of UTF-8, the first byte in the sequence can represent either a literal value or a format-indicator that specifies the number of bytes to follow. M32 also resembles the variable-length integer code used by SQLite4 (SQLite4, 2019), except that it supports negative values as well as positive.

The first byte in an M32-seriealized integer is unusual with respect to the GVRS specification. First, it is treated as a signed integral value. In GVRS, bytes are almost always treated as unsigned values. The first byte is also essentially a hybrid. For small-magnitude integer symbols (in the range -126 to +126), it serves to represent the signed integer value of the symbol (resulting in a M32 serialization of length 1). The values -127 and +127 are used to introduce multi-byte sequences (-127 is used for negative-valued integer symbols, +127 is used for positive-valued symbols). Finally, the value -128 is used to indicate the minimum 32-bit integer value, -2147483648, which GVRS often uses as a null-data code. In the multi-byte sequences, the bytes that follow the introducer give bits for the numerical value in big-endian order. Each byte carries 7 bits of information with the high-order bit used to indicate if additional bytes follow. The length of the sequence depends on the magnitude of the residual to be coded. In the worst case, 6 bytes are required to encode large magnitude residual (1 byte for the introducer, 5 bytes for the content).

The M32 coding scheme is effective in the case where the computed residuals tend to be small because it preserves the one-byte-per-value relationship. Again, in the case of larger residuals, the M32 code can actually be longer than the 4 bytes needed for a simple integer. Fortunately, a good predictor will usually produce small residuals.

7.1 A definition for the M32 coding sequence

The M32 serializes a four-byte signed integer value into a sequence of 1 to 6 bytes according to the following rules.

1. Integer values in the range -126 to +126 are stored as a single signed byte.
2. Integer values -2147483648 (minimum integer value) receive special treatment and are stored using the signed byte code -128 (unsigned 255).
3. If an integer value is less than or equal to -127, the first byte in the output sequence is set to negative 127 and the absolute value of the integer is stored in subsequent bytes following the rules below.
4. If an integer value is greater than or equal to 127, the first by in the sequence is set to 127 and the value of the integer is stored in subsequent bytes according to the rules given below.

Storage format for values or absolute values greater than or equal to 127:

1. The integer is stored in a sequence of bytes. Each byte carries 7 bits of data extracted from the input and 1 bit of data indicating whether additional bytes follow or if the

Storage for integers with an absolute value in the range [0, 126]

Byte	Content
0	Signed by representation of integer value

Stage for integers with an absolute value in the range [127, 254], 14 bits of storage

Byte	Content
0	127 for position values, -127 for negative

Byte	High bit	Low seven bits
1	1	Bits 7 to 13 from input
2	0	Bits 0 to 6 from input

Stage for integers with an absolute value in the range [255, 16638], 21 bits of storage

Byte	Content
0	127 for position values, -127 for negative

Byte	High bit	Low seven bits
1	1	Bits 14 to 20 from input
2	1	Bits 7 to 13 from input
3	0	Bits 0 to 6 from input

Stage for integers with an absolute value in the range [16639, 2113790], 28 bits of storage

Byte	Content
0	127 for position values, -127 for negative

Byte	High bit	Low seven bits
1	1	Bits 21 to 27 from input
2	1	Bits 14 to 20 from input
3	1	Bits 7 to 13 from input
4	0	Bits 0 to 6 from input

Storage for integers with an absolute value in the range [270549247, 2147483647], 31 bits of storage

Byte	Content
0	127 for position values, -127 for negative

Byte	High bit	Low seven bits
1	1	Bits 28 to 31 from input
2	1	Bits 21 to 27 from input
3	1	Bits 14 to 20 from input
4	1	Bits 7 to 13 from input
5	0	Bits 0 to 6 from input

Special case storage for integers with the value -2147483648 (minimum value for a signed 32 bit integer)

Byte	Content
0	-128

The choice to store integer symbols as absolute values rather than in two's compliment form is based on test results with elevation data. Tests with different formats indicated a small improvement in the redundancy of the encoded symbols when absolute values were used. Similar tests showed an improvement in compression ratios when symbols were stored with their higher-order bits given first.

The design decision to reserve the byte code of -128 for the value -2147483648 (minimum value for a signed 32 bit integer) was based on the expectation that integer data sets will often reserve that value to represent null-data cells. In cases where tiles feature a mix of standard and null-data cells, having a special code for null values can improve overall data compression.

8 Bit sequence storage and ordering

The Huffman data compression encodes data a bit at a time. GVRS specifies the following rules for packing a sequence of bits into a collection of bytes.

The following rules specify the storage of a single bit to a collection of bytes.

1. Incoming bits are stored in the "current byte" in the collection.
2. Bits are stored in increasing order of bit position within the current byte
    a. The first bit is stored in the low-order bit position of the current byte (bit zero)
    b. The next bit is stored in bit one of the current byte.
    c. The process continues with increasing storage bit position until 8 bits are stored in the current byte (e.g. the current byte is "full").
3. When the current byte is full (contains 8 bits), the next byte in the collection is defined as the current byte.

Some applications (including the Huffman code) require the storage of integer values larger than one byte. In this case, the rules for storing a single bit are extended as follows:

1. If the value to be stored can be fit within the remaining unclaimed bits in the current byte, the value is stored in the unclaimed (high-order) section of the current byte.
2. If the value is too large to fit within the unclaimed space:
    a. The low-order bits of the value will be stored in the unclaimed space.
    b. The value will be shifted down by the number of bits stored.
    c. The "current byte" will be advanced through the byte collection
    d. The process will repeat from step 1 until all bits in the input value have been processed.

9 Grid index and real-valued coordinate systems

For applications that do not require geometry-based operations, a raster data set can be viewed as a simple two-dimensional array. In such cases, the data contained in the raster product can be accessed using integer indices. The figure below illustrates a typical layout of a non-spatial data grid. The variable *i* is used to indicate row and *j* to indicate column. Variables $z_{i,j}$ give the values for each data cell. For reasons of software efficiency, the Gridfour library often uses arrays of one-dimension rather than two. In one-dimensional arrays, the data is organized in row-major order. The small numbers in the upper-left corner of each data cell in the figure represent the order of elements within the raster grid. They correspond to the integer indices for array elements when the grid is stored in a one-dimensional array. This ordering scheme is also used when data is serialized for storage or transmission between applications.

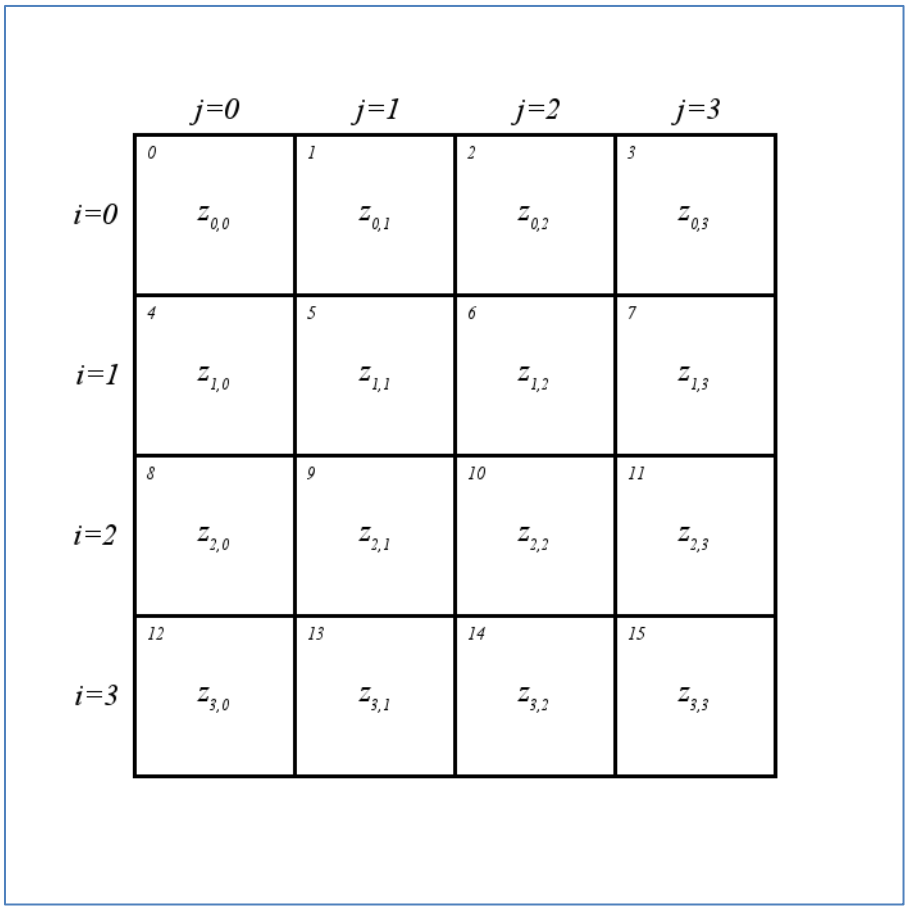


Figure 4 – Integer indices for a raster.

9.1 Real-valued coordinates for spatial data

Many applications apply a spatial interpretation to raster data sets. In such cases, we extend the variables *i* and *j* to real-valued counterparts, *i'* and *j'*. These coordinates, which we refer to as "grid coordinates", allow us to indicate any point within the raster grid using unitless, real-valued coordinates based on row and column. Gridfour also allows applications to tie real-world coordinates (Cartesian coordinates, geographic coordinates, etc.) to the grid. To do so, an "anchor point" designated (x_0, y_0) is attached to the center of the cell in the first row and first column of the grid. Cells are treated as having a uniform width, *w*, and height, *h*.

The figure below shows an example of the assignments of coordinates to a raster product with a spatial basis.

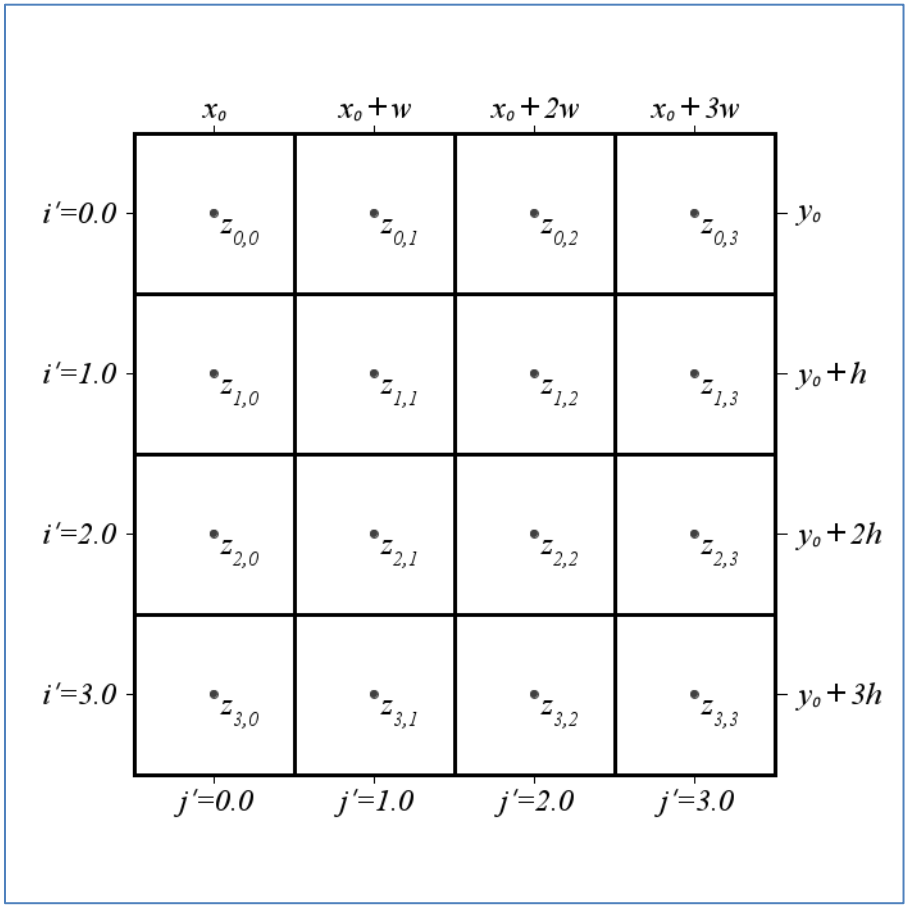


Figure 5 – Real-valued coordinates for a grid with a spatial basis.

9.1.1 The center-point sample interpretation of data values

The figure above shows points at the center of each raster cell. The Gridfour software library often (but not always) assumes that the data in a raster product with a spatial basis represents a continuous surface. The data values are treated as being tied to points located at the center of each grid cell. These points are assigned integral coordinates. Points falling between data samples are assigned non-integral coordinates.

The center-point interpretation of data values is useful for interpolation, data-smoothing, digital filtering, and other applications that perform operations over a continuous surface.

9.1.2 Real-valued coordinate systems defined for the GVRS format

The GVRS format currently defines two broad categories of real-valued coordinate systems:

> **Grid Coordinates** are the row and column specifications *i'* and *j'* that were described above. Grid coordinates are used to perform lookup operations on raster data sets, and are also applied to interpolation operations.
>
> **Model Coordinates** are real-valued coordinates computed using information about the cell-size and anchor position described above. Currently, GVRS supports two subcategories for model coordinates: Cartesian *(x, y)* and geographic coordinates (*latitude* and *longitude*). Additional kinds of model coordinates (including polar and complex coordinates) may be considered in the future.

The GVRS format provides specifications about the anchor point and cell sizes to establish *affine transforms* for mapping model coordinates to grid coordinates. The API also allows applications to specify custom affine transforms to support operations such as rotations, skewed axes, or reflection

The model-to-raster (grid) coordinate transform is specified in the file header using elements M2R00, M2R01, etc. (see paragraph 0).

$$\begin{vmatrix} M2R00 & M2R01 & M2R02 \\ M2R10 & M2R11 & M2R12 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} x \\ y \\ 1 \end{vmatrix} = \begin{vmatrix} column \\ row \\ 1 \end{vmatrix}$$

The raster-to-model coordinate transform is specified in the file header using elements R2M00, R2M01, etc.

$$\begin{vmatrix} R2M00 & R2M01 & R2M02 \\ R2M10 & R2M11 & R2M12 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} column \\ row \\ 1 \end{vmatrix} = \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

10 Metadata naming and data type conventions

The name, record ID, and data type specifications for a GVRS file are strictly under the control of the application that creates it. There is, however, an advantage to different packaging programs following a consistent standard where it makes sense to do so. In view of that, the Gridfour project created a list of pre-defined conventions for naming certain kinds of metadata records. Application developers are encouraged to follow these conventions. If you encounter additional naming conventions that would be a useful addition to the list, please contact the Gridfour project or post a notice in the project's Discussions or Issues pages.

The Java reference implementation includes an enumerated type that embodies these conventions. See GvrsMetadataNames.java for more information.

Note that some of the conventions listed below may describe elements that may have multiple entries. For example, a document may have multiple authors. In that case, a GVRS file could include multiple metadata entries with the name "Author" but different record IDs.

Name	Data type	Description
Author	String	The person or organization that created a data product.
Copyright	String	Copyright notice, when required by law.
TermsOfUse	String	A statement giving conditions under which the producer released the associated product for use. May indicating restrictions on use, limitations of applicability, etc. For example: "Not intended for navigation".
Disclaimers	String	A statement disclaiming liability or clarifying limitations of applicability for the product.
TIFF	Unspecified	Defines a specification for bundling "tag" elements from the industry standard Tagged Image File Format (TIFF) into a GVRS file. By convention, the record ID used when creating metadata for TIFF specifications should be the integer TIFF tag ID. TIFF tags may include data of various types, thus the GVRS convention does not specify any particular type for this metadata.
WKT	String	Well-Known Text (WKT) is a standard used for Geographic Information Systems (GIS) to provide coordinate system, map-projection, and related information.
GvrsCompressionCodecs	ASCII	A simple pipe-delimited list indicating which data-compression codecs were used for a GVRS file
GvrsJavaCodecs	ASCII	A list giving the GVRS compression codec name and the associated Java classpath for the encoder and decoder classes. Suitable for Java implementations.

11 References

Castagnoli, G., Brauer, S., & Herrmann, M., "Optimization of cyclic redundancy-check codes with 24 and 32 parity bits," in *IEEE Transactions on Communications*, vol. 41, no. 6, pp. 883-892, June 1993, doi: 10.1109/26.231911.

Deutsch, L. Peter (1996). "DEFLATE Compressed Data Format Specification version 1.3". IETF. p. 1. sec. Abstract. doi:10.17487/RFC1951. RFC 1951.

Kidner, D.B. & Smith, D.H. (1992). "Compression of digital elevation models by Huffman coding". Computers and Geosciences, 18(8), 1013-1034.

Leach, P. (2005). “A Universally Unique Identifier (UUID) URN Namespace”, Request for Comment (RFC) 2122, IETF Network Working Group. Accessed September 2022 from https://www.ietf.org/rfc/rfc4122.txt

SQLite4 (2019). "Variable-Length Integers". Retrieved October 2019 from https://sqlite.org/src4/doc/trunk/www/varint.wiki.

Sullivan, Louis H. (1896). “The Tall Office Building Artistically Considered”, Lippincott’s Magazine (March 1896), p. 403-409.