Viewing Binary Files with the hexView Package

by Paul Murrell

I really like plain text files.

I like them because I can see exactly what is in them. I can even easily modify the file if I'm feeling dangerous. This makes me feel like I understand the file.

I am not so fond of binary files. I always have to use specific software to access the contents and that software only shows me an interpretation of the basic content of the file. The raw content is hidden from me.

Sometimes I want to know more about a real binary file, for example when I need to read data in a binary format that no existing R function will read. When things go wrong, like when an R workspace file becomes "corrupt", I may have a strong *need* to know more.

Hex editors are wonderful tools that provide a view of the raw contents of a binary (or text) file, whether just to aid in understanding the file or to inspect or recover a file. The **hexView** package is an attempt to bring this sort of facility to R.

Viewing raw text files

The `viewRaw()` function reads and displays the raw content of a file. The content is displayed in three columns: the left column provides a byte offset within the file, the middle column shows the raw bytes, and the right column displays each byte as an ASCII character. If the byte does not correspond to a printable ASCII character then a full stop is displayed.

As a simple example, we will look at a plain text file, `"rawTest.txt"`, that contains a single line of text. This file was created using the following code (on a Linux system).

```
> writeLines("test pattern", "rawTest.txt")
```

A number of small example files are included as part of the **hexView** package and the `hexViewFile()` function is provided to make it convenient to refer to these files. The `readLines()` function from the **base** package reads in the lines of a plain text file as a vector of strings, so the plain text content of the file `"rawTest.txt"` can be retrieved as follows.

```
> readLines(hexViewFile("rawTest.txt"))

[1] "test pattern"
```

The following code uses the `viewRaw()` function from **hexView** to display the *raw* contents of this file.

```
> viewRaw(hexViewFile("rawTest.txt"))

0  :  74 65 73 74 20 70 61 74  |  test pat
8  :  74 65 72 6e 0a           |  tern.
```

As this example shows, by default, the raw bytes are printed in hexadecimal format. The first byte in this file is `74`, which is $7 * 16 + 4 = 116$ in decimal notation—the ASCII code for the character `t`. This byte pattern can be seen several times in the file, wherever there is a `t` character.

The `machine` argument to the `viewRaw()` function controls how the raw bytes are displayed. It defaults to `"hex"` for hexadecimal output, but also accepts the value `"binary"`, which means that the raw bytes are printed in binary format, as shown below.

```
> viewRaw(hexViewFile("rawTest.txt"),
          machine="binary")

 0  :  01110100 01100101 01110011  |  tes
 3  :  01110100 00100000 01110000  |  t p
 6  :  01100001 01110100 01110100  |  att
 9  :  01100101 01110010 01101110  |  ern
12  :  00001010                    |  .
```

One noteworthy feature of this simple file is the last byte, which has the hexadecimal value `0a` (or `00001010` in binary; the decimal value `10`) and no printable ASCII interpretation. This is the ASCII code for the *newline* or *line feed* (LF) special character that indicates the end of a line in text files. This is a simple demonstration that even plain text files have details that are hidden from the user by standard viewing software; viewers will show text on separate lines, but do not usually show the "character" representing the start of a new line.

The next example provides a more dramatic demonstration of hidden details in text files. The file we will look at contains the same text as the previous example, but was created on a Windows XP system with Notepad using "Save As..." and selecting "Unicode" as the "Encoding". The `readLines()` function just needs the file to be opened with the appropriate encoding, then it produces the same result as before.

```
> readLines(
      file(hexViewFile("rawTest.unicode"),
           encoding="UCS-2LE"))

[1] "test pattern"
```

However, the raw content of the file is now very different.

```
> viewRaw(hexViewFile("rawTest.unicode"))
```

```
 0  :  ff fe 74 00 65 00 73 00   |  ..t.e.s.
 8  :  74 00 20 00 70 00 61 00   |  t. .p.a.
16  :  74 00 74 00 65 00 72 00   |  t.t.e.r.
24  :  6e 00 0d 00 0a 00         |  n.....
```

It is fairly straightforward to identify some parts of this file. The ASCII codes from the previous example are there again, but there is an extra 00 byte after each one. This reflects the fact that, on Windows, Unicode text is stored using two bytes per character[1].

Instead of the 13 bytes in the original file, we might expect 26 bytes in this file, but there are actually 30 bytes. Where did the extra bytes come from?

The first two bytes at the start of the file are a *byte order mark* (BOM). With two bytes to store for each character, there are two possible orderings of the bytes; for example, the two bytes for the character t could be stored as 74 00 (called *little endian*) or as 00 74 (*big endian*). The BOM tells software which order has been used. Another difference occurs at the end of the file. The newline character is there again (with an extra 00), but just before it there is a 0d character (with an extra 00). This is the *carriage return* (CR) character. On Windows, a new line in a text file is signalled by the combination CR+LF, but on UNIX a new line is just indicated by a single LF.

As this example makes clear, software sometimes does a lot of work behind the scenes in order to display even “plain text”.

Viewing raw binary files

An example of a binary file is the native binary format used by R to store information via the save() function. The following code was used to create the file "rawTest.bin".

```
> save(rnorm(50), file="rawTest.bin")
```

We can view this file with the following code; the nbytes argument is used to show the raw data for only the first 80 bytes.

```
> viewRaw(hexViewFile("rawTest.bin"),
          nbytes=80)
```

```
 0  :  1f 8b 08 00 00 00 00 00   |  ........
 8  :  00 03 01 c0 01 3f fe 52   |  .....?.R
16  :  44 58 32 0a 58 0a 00 00   |  DX2.X...
24  :  00 02 00 02 05 00 00 02   |  ........
32  :  03 00 00 00 04 02 00 00   |  ........
40  :  00 01 00 00 10 09 00 00   |  ........
48  :  00 01 7a 00 00 00 0e 00   |  ..z.....
56  :  00 00 32 3f e7 60 e6 49   |  ..2?.`.I
64  :  c6 fe 0d 3f e1 3b c5 2f   |  ...?.;./
72  :  bb 4e 18 bf c4 9e 0f 1a   |  .N......
```

This is a good example of a binary file that is intriguing to view, but there is little hope of retrieving any useful information because the data has been compressed (encoded). In other cases, things are a not so hopeless, and it is not only possible to view the raw bytes, but also to see useful patterns and structures.

The next example looks at a binary file with a much simpler structure. The file "rawTest.int" only contains (uncompressed) integer values and was created by the following code.

```
> writeBin(1:50, "rawTest.int", size=4)
```

This file only contains the integers from 1 to 50, with four bytes used for each integer. The raw contents are shown below; this time the nbytes argument has been used to show only the raw data for the first 10 integers (the first 40 bytes).

```
> viewRaw(hexViewFile("rawTest.int"),
          nbytes=40)
```

```
 0  :  01 00 00 00 02 00 00 00   |  ........
 8  :  03 00 00 00 04 00 00 00   |  ........
16  :  05 00 00 00 06 00 00 00   |  ........
24  :  07 00 00 00 08 00 00 00   |  ........
32  :  09 00 00 00 0a 00 00 00   |  ........
```

None of the bytes correspond to printable ASCII characters in this case, so the right column of output is not terribly interesting. The viewRaw() function has two arguments, human and size, which control the way that the raw bytes are interpreted and displayed. In this case, rather than interpreting each byte as an ASCII character, it makes sense to interpret each block of four bytes as an integer. This is done in the following code using human="int" and size=4.

```
> viewRaw(hexViewFile("rawTest.int"),
          nbytes=40, human="int", size=4)
```

```
 0  :  01 00 00 00 02 00 00 00   |   1  2
 8  :  03 00 00 00 04 00 00 00   |   3  4
16  :  05 00 00 00 06 00 00 00   |   5  6
24  :  07 00 00 00 08 00 00 00   |   7  8
32  :  09 00 00 00 0a 00 00 00   |   9 10
```

With this simple binary format, we can see how the individual integers are being stored. The integer 1 is stored as the four bytes 01 00 00 00, the integer 2 as 02 00 00 00, and so on. This clearly demonstrates the idea of little endian byte order; the least-significant byte, the value 1, is stored first. In big endian byte order, the integer 1 would be 00 00 00 01 (as we shall see later).

The other option for interpreting bytes is "real" which means that each block of size bytes is interpreted as a floating-point value. A simple example

[1]Over-simplification alert! Windows *used to* use the UCS-2 encoding, which has two bytes per character, but now it uses UTF-16, which has two or four bytes per character. There are only two bytes per character in this case because these are common english characters.

is provided by the file "rawTest.real", which was generated by the following code. I have deliberately used big endian byte order because it will make it easier to see the structure in the resulting bytes.

```
> writeBin(1:50/50, "rawTest.real", size=8,
           endian="big")
```

Here is an example of reading this file and interpreting each block of 8 bytes as a floating-point number. This also demonstrates the use of the width argument to explicitly control how many bytes are displayed per line of output.

```
> viewRaw(hexViewFile("rawTest.real"),
          nbytes=40, human="real", width=8,
          endian="big")
```

```
 0  :  3f 94 7a e1 47 ae 14 7b  |  0.02
 8  :  3f a4 7a e1 47 ae 14 7b  |  0.04
16  :  3f ae b8 51 eb 85 1e b8  |  0.06
24  :  3f b4 7a e1 47 ae 14 7b  |  0.08
32  :  3f b9 99 99 99 99 99 9a  |  0.10
```

Again, we are able to see how individual floating-point values are stored. The following code takes this a little further and allows us to inspect the bit representation of the floating point numbers. The output is shown in Figure 1.

```
> viewRaw(hexViewFile("rawTest.real"),
          nbytes=40, human="real",
          machine="binary", width=8,
          endian="big")
```

The bit representation adheres to the IEEE Standard for Binary Floating-Point Arithmetic (IEEE, 1985; Wikipedia, 2006). Each value is stored in the form $sign \times mantissa \times 2^{exponent}$. The first (left-most) bit indicates the sign of the number, the next 11 bits describe the *exponent* and the remaining 52 bits describe the *mantissa*. The mantissa is a binary fraction, with bit i corresponding to 2^{-i}.

For the first value in "rawTest.real", the first bit has value 0 indicating a positive number, the exponent bits are 0111111 1001 = 1017, from which we subtract 1023 to get -6, and the mantissa is an implicit 1 plus $0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + 1 \times 2^{-6} ... = 1.28$.[2] So we have the value $1.28 \times 2^{-6} = 0.02$.

Viewing a Binary File in Blocks

As the examples so far have hopefully demonstrated, being able to see the raw contents of a file can be a very good way to teach concepts such as endianness, character encodings, and floating-point representations of real numbers. Plus, it is just good fun to poke around in a file and see what is going on.

In this section, we will look at some more advanced functions from the **hexView** package, which will allow us to take a more detailed look at more complex binary formats and will allow us to perform some more practical tasks.

We will start by looking again at R's native binary format. The file "rawTest.XDRint" contains the integers 1 to 50 saved as a binary R object and was produced using the following code. The compress=FALSE is important to allow us to see the structure of the file.

```
> save(1:50, file="rawTest.XDRint",
       compress=FALSE)
```

We can view (the first 80 bytes of) the raw file using viewRaw() as before and this does show us some interesting features. For example, we can see the text RDX2 at the start of the file (it is common for files to have identifying markers at the start of the file). If we look a little harder, we can also see the first few integers (1 to 9); the data is stored in an XDR format (Wikipedia, 2006a), which uses big endian byte order, so the integers are in consecutive blocks of four bytes that look like this: 00 00 00 01, then 00 00 00 02, and so on.

```
> viewRaw(hexViewFile("rawTest.XDRint"),
          width=8, nbytes=80)
```

```
 0  :  52 44 58 32 0a 58 0a 00  |  RDX2.X..
 8  :  00 00 02 00 02 04 00 00  |  ........
16  :  02 03 00 00 00 04 02 00  |  ........
24  :  00 00 01 00 00 10 09 00  |  ........
32  :  00 00 01 78 00 00 00 0d  |  ...x....
40  :  00 00 00 32 00 00 00 01  |  ...2....
48  :  00 00 00 02 00 00 00 03  |  ........
56  :  00 00 00 04 00 00 00 05  |  ........
64  :  00 00 00 06 00 00 00 07  |  ........
72  :  00 00 00 08 00 00 00 09  |  ........
```

It is clear that there is some text in the file and that there are some integers in the file, so neither viewing the whole file as characters nor viewing the whole file as integers is satisfactory. What we need to be able to do is view the text sections as characters and the integer sections as integers. This is what the functions memFormat(), memBlock(), and friends are for.

The memBlock() function creates a description of a block of memory, specifying how many bytes are in the block; the block is interpreted as ASCII characters. The atomicBlock() function creates a description of a memory block that contains a single value of a specified type (e.g., a four-byte integer), and the vectorBlock() function creates a description of a memory block consisting of 1 or more memory blocks.

A number of standard memory blocks are predefined: integer4 (a four-byte integer) and integer1,

[2] At least, as close as it is possible to get to 1.28 with a finite number of bits. Another useful thing about viewing raw values is that it makes explicit the fact that most decimal values do not have an exact floating-point representation.

```
 0  :  00111111 10010100 01111010 11100001 01000111 10101110 00010100 01111011  |  0.02
 8  :  00111111 10100100 01111010 11100001 01000111 10101110 00010100 01111011  |  0.04
16  :  00111111 10101110 10111000 01010001 11101011 10000101 00011110 10111000  |  0.06
24  :  00111111 10110100 01111010 11100001 01000111 10101110 00010100 01111011  |  0.08
32  :  00111111 10111001 10011001 10011001 10011001 10011001 10011001 10011010  |  0.10
```

Figure 1: The floating point representation of the numbers 0.02 to 0.10 following IEEE 754 in big endian byte order.

integer2, and integer8; real8 (an eight-byte floating-point number, or *double*) and real4; and ASCIIchar (a single-byte character). There is also a special ASCIIline memory block for a series of single-byte characters terminated by a newline.

The memFormat() function collects a number of memory block descriptions together and viewFormat() reads the memory blocks and displays them.

As an example, the following code reads in the "RDX2" header line of the file "rawTest.XDRint", treats the next 39 bytes as just raw binary, ignoring any structure, then reads the first nine integers (as integers). A new memory block description is needed for the integers because the XDR format is big endian (the predefined integer4 is little endian). The names of the memory blocks within the format are used to separate the blocks of output.

```
> XDRint <- atomicBlock("int", endian="big")
> viewFormat(hexViewFile("rawTest.XDRint"),
      memFormat(saveFormat=ASCIIline,
                rawBlock=memBlock(39),
                integers=vectorBlock(XDRint,
                                     9)))
```

```
========saveFormat
  0  :  52 44 58 32 0a           |  RDX2.
========rawBlock
  5  :  58 0a 00 00 00 02 00     |  X......
 12  :  02 04 00 00 02 03 00     |  .......
 19  :  00 00 04 02 00 00 00     |  .......
 26  :  01 00 00 10 09 00 00     |  .......
 33  :  00 01 78 00 00 00 0d     |  ..x....
 40  :  00 00 00 32              |  ...2
========integers
 44  :  00 00 00 01 00 00 00 02  |  1 2
 52  :  00 00 00 03 00 00 00 04  |  3 4
 60  :  00 00 00 05 00 00 00 06  |  5 6
 68  :  00 00 00 07 00 00 00 08  |  7 8
 76  :  00 00 00 09              |  9
```

The raw 39 bytes can be further broken down—see the description of R's native binary format on pages 11 and 12 of the "R Internals" manual (R Development Core Team, 2006) that is distributed with R—but that is beyond the scope of this article.

Extracting Blocks from a Binary File

As well as viewing different blocks of a binary file, we may want to extract the values from each block. For this purpose, the readFormat() function is provided to read a binary format, as produced by memFormat(), and generate a "rawFormat" object (but not explicitly print it[3]). A "rawFormat" object is a list with a component "blocks" that is itself a list of "rawBlock" objects, one for each memory block defined in the memory format. A "rawBlock" object contains the raw bytes read from a file.

The blockValue() function extracts the interpreted value from a "rawBlock" object. The blockString() function is provided specifically for extracting a null-terminated string from a "rawBlock" object.

The following code reads in the file "rawTest.XDRint" and just extracts the 50 integer values.

```
> XDRfile <-
    readFormat(hexViewFile("rawTest.XDRint"),
      memFormat(saveFormat=ASCIIline,
                rawBlock=memBlock(39),
                integers=vectorBlock(XDRint,
                                     50)))
> blockValue(XDRfile$blocks$integers)

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
[14] 14 15 16 17 18 19 20 21 22 23 24 25 26
[27] 27 28 29 30 31 32 33 34 35 36 37 38 39
[40] 40 41 42 43 44 45 46 47 48 49 50
```

A Caution

On a typical 32-bit platform, R uses 4 bytes for representing integer values in memory and 8 bytes for floating-point values. This means that there may be limits on what sort of values can be *interpreted* correctly by **hexView**.

For example, if a file contains 8-byte integers, it is possible to view each set of 8 bytes as an integer, but on my system R can only represent an integer using 4 bytes, so 4 of the bytes are (silently) dropped. The following code demonstrates this effect by reading

[3]There is a readRaw() function too.

the file "testRaw.int" and interpreting its contents as 8-byte integers.

```
> viewRaw(hexViewFile("rawTest.int"),
          nbytes=40, human="int", size=8)

 0  :  01 00 00 00 02 00 00 00  |  1
 8  :  03 00 00 00 04 00 00 00  |  3
16  :  05 00 00 00 06 00 00 00  |  5
24  :  07 00 00 00 08 00 00 00  |  7
32  :  09 00 00 00 0a 00 00 00  |  9
```

An extended example: Reading EViews Files

On November 18 2006, Dietrich Trenkler sent a message to the R-help mailing list asking for a function to read files in the native binary format used by Eviews, an econometrics software package (http://www.eviews.com/). No such function exists, but John C Frain helpfully pointed out that an unofficial description of the basic structure of Eviews files had been made available by Allin Cottrell (creator of gretl, the Gnu Regression, Econometrics and Time-series Library). The details of Allin Cottrell's reverse-engineering efforts are available on the web (http://www.ecn.wfu.edu/~cottrell/eviews_format/).

In this section, we will use the **hexView** package to explore an Eviews file and produce a new function for reading files in this format. The example data file we will use is from Ramu Ramanathan's Introductory Econometrics text (Ramanathan, 2002). The data consists of four variables measured on single family homes in University City, San Diego, in 1990:

price: sale price in thousands of dollars.

sqft: square feet of living area.

bedrms: number of bedrooms.

baths: number of bath rooms.

The data are included in both plain text format, as "data4-1.txt", and Eviews format, as "data4-1.wf1", as part of the **hexViews** package.[4] For later comparison, the data from the plain text format are shown below, having been read in with the read.table() function.

```
> read.table(hexViewFile("data4-1.txt"),
             col.names=c("price", "sqft",
               "bedrms", "baths"))

   price sqft bedrms baths
1  199.9 1065      3  1.75
2  228.0 1254      3  2.00
3  235.0 1300      3  2.00
4  285.0 1577      4  2.50
5  239.0 1600      3  2.00
6  293.0 1750      4  2.00
7  285.0 1800      4  2.75
8  365.0 1870      4  2.00
9  295.0 1935      4  2.50
10 290.0 1948      4  2.00
11 385.0 2254      4  3.00
12 505.0 2600      3  2.50
13 425.0 2800      4  3.00
14 415.0 3000      4  3.00
```

An Eviews file begins with a header, starting with the text "New MicroTSP Workfile" and including important information about the size of the header and the number of variables and the number of observations in the file. The following code defines an appropriate "memFormat" object for this header information.

```
> EViewsHeader <-
    memFormat(firstline=memBlock(80),
              headersize=integer8,
              unknown=memBlock(26),
              numvblesplusone=integer4,
              date=vectorBlock(ASCIIchar, 4),
              unkown=memBlock(2),
              datafreq=integer2,
              startperiod=integer2,
              startobs=integer4,
              unkown=memBlock(8),
              numobs=integer4)
```

We can use readFormat() to read this header from the file as follows. The number of variables reported is one greater than the actual number of variables and also includes two "boiler plate" variables that are always included in Eviews files (hence 7 instead of the expected 4).

```
> data4.1.header <-
     readFormat(hexViewFile("data4-1.wf1"),
                EViewsHeader)
> data4.1.header

=========firstline
  0  :  4e 65 77 20 4d 69                    |  New Mi
  6  :  63 72 6f 54 53 50                    |  croTSP
 12  :  20 57 6f 72 6b 66                    |   Workf
 18  :  69 6c 65 00 00 00                    |  ile...
 24  :  d8 5e 0e 01 00 00                    |  .^....
 30  :  00 00 00 00 08 00                    |  ......
 36  :  15 00 00 00 00 00                    |  ......
 42  :  ff ff ff ff 21 00                    |  ....!.
 48  :  00 00 00 00 00 00                    |  ......
 54  :  00 00 06 00 00 00                    |  ......
 60  :  0f 00 00 00 06 00                    |  ......
 66  :  00 00 01 00 01 00                    |  ......
 72  :  66 03 00 00 00 00                    |  f.....
 78  :  00 00                                |  ..
=========headersize
 80  :  90 00 00 00 00 00 00 00             |  144
=========unknown
 88  :  01 00 00 00 01 00                    |  ......
```

[4]The original source of the files was: http://ricardo.ecn.wfu.edu/pub/gretl_cdrom/data/

```
  94  :  00 00 01 00 00 00                  |  ......
 100  :  00 00 00 00 00 00                  |  ......
 106  :  00 00 00 00 00 00                  |  ......
 112  :  00 00                              |  ..
=========numvblesplusone
 114  :  07 00 00 00                        |  7
=========date
 118  :  d5 b7 0d 3a                        |  ...:
=========unkown
 122  :  06 00                              |  ..
=========datafreq
 124  :  01 00                              |  1
=========startperiod
 126  :  00 00                              |  0
=========startobs
 128  :  01 00 00 00                        |  1
=========unkown
 132  :  00 5d 67 0e 01 59                  |  .]g..Y
 138  :  8b 41                              |  .A
=========numobs
 140  :  0e 00 00 00                        |  14
```

We can extract some pieces of information from this header and use them to look at later parts of the file.

```
> headerSize <-
      blockValue(
          data4.1.header$blocks$headersize)
> numObs <-
      blockValue(
          data4.1.header$blocks$numobs)
```

At a location 26 bytes beyond the header size, there are several blocks describing each variable in the Eviews file. Each of these blocks is 70 bytes long and contains information on the variable name and the location within the file where the data values reside for that variable. The following code creates a description of a block containing variable information, then uses `readFormat()` to read the information for the first variable (the number of bath rooms); the `offset` argument is used to start reading the block at the appropriate location within the file. We also extract the location of the data for this variable.

```
> EViewsVbleInfo <-
    memFormat(unknown=memBlock(6),
              recsize=integer4,
              memsize=integer4,
              ptrtodata=integer8,
              vblename=vectorBlock(ASCIIchar,
                32),
              ptrtohistory=integer8,
              vbletype=integer2,
              unknown=memBlock(6))
> data4.1.vinfo <-
      readFormat(hexViewFile("data4-1.wf1"),
                 EViewsVbleInfo,
                 offset=headerSize + 26)
> data4.1.vinfo
```

```
=========unknown
 170  :  00 00 00 00 0b 00              |  ......
=========recsize
 176  :  86 00 00 00                     |  134
=========memsize
 180  :  70 00 00 00                     |  112
=========ptrtodata
 184  :  f6 03 00 00 00 00 00 00         |  1014
=========vblename
 192  :  42 41 54 48 53 00 00 00 00 00   |  BATHS.....
 202  :  00 00 00 00 00 00 00 00 00 00   |  ..........
 212  :  00 00 00 00 00 00 00 00 00 00   |  ..........
 222  :  00 00                           |  ..
=========ptrtohistory
 224  :  00 00 00 00 d5 b7 0d 3a         |  0
=========vbletype
 232  :  2c 00                           |  44
=========unknown
 234  :  60 02 10 00 01 00               |  `.....
```

```
> firstVbleLoc <-
    blockValue(data4.1.vinfo$blocks$ptrtodata)
```

The data for each variable is stored in a block containing some preliminary information followed by the data values stored as eight-byte floating-point numbers. The code below creates a description of a block of variable data and then reads the data block for the first variable.

```
> EViewsVbleData <- function(numObs) {
      memFormat(numobs=integer4,
                startobs=integer4,
                unknown=memBlock(8),
                endobs=integer4,
                unknown=memBlock(2),
                values=vectorBlock(real8,
                  numObs))
  }
> viewFormat(hexViewFile("data4-1.wf1"),
             EViewsVbleData(numObs),
             offset=firstVbleLoc)
```

```
=========numobs
1014  :  0e 00 00 00                        |  14
=========startobs
1018  :  01 00 00 00                        |  1
=========unknown
1022  :  00 00 00 00 00 00                  |  ......
1028  :  00 00                              |  ..
=========endobs
1030  :  0e 00 00 00                        |  14
=========unknown
1034  :  00 00                              |  ..
=========values
1036  :  00 00 00 00 00 00 fc 3f            |  1.75
1044  :  00 00 00 00 00 00 00 40            |  2.00
1052  :  00 00 00 00 00 00 00 40            |  2.00
1060  :  00 00 00 00 00 00 04 40            |  2.50
1068  :  00 00 00 00 00 00 00 40            |  2.00
1076  :  00 00 00 00 00 00 00 40            |  2.00
1084  :  00 00 00 00 00 00 06 40            |  2.75
1092  :  00 00 00 00 00 00 00 40            |  2.00
1100  :  00 00 00 00 00 00 04 40            |  2.50
1108  :  00 00 00 00 00 00 00 40            |  2.00
1116  :  00 00 00 00 00 00 08 40            |  3.00
1124  :  00 00 00 00 00 00 04 40            |  2.50
1132  :  00 00 00 00 00 00 08 40            |  3.00
1140  :  00 00 00 00 00 00 08 40            |  3.00
```

This manual process of exploring the file structure can easily be automated within a function. The **hexView** package includes such a function under the name `readEViews()`. With this function, we can read in the data set from the Eviews file as follows.

```
> readEViews(hexViewFile("data4-1.wf1"))
```

```
Skipping boilerplate variable
Skipping boilerplate variable
   BATHS BEDRMS PRICE SQFT
1   1.75      3 199.9 1065
2   2.00      3 228.0 1254
3   2.00      3 235.0 1300
4   2.50      4 285.0 1577
5   2.00      3 239.0 1600
6   2.00      4 293.0 1750
7   2.75      4 285.0 1800
8   2.00      4 365.0 1870
9   2.50      4 295.0 1935
10  2.00      4 290.0 1948
11  3.00      4 385.0 2254
12  2.50      3 505.0 2600
13  3.00      4 425.0 2800
14  3.00      4 415.0 3000
```

This solution is not the most efficient way to read Eviews files, but the **hexView** package does make it easy to gradually build up a solution, it makes it easy to view the results, and it does provide a way to solve the problem without having to resort to C code.

Summary

The **hexView** package provides functions for viewing the raw byte contents of files. This is useful for exploring a file structure and for demonstrating how information is stored on a computer. More advanced functions make it possible to read quite complex binary formats using only R code.

Acknowledgements

At the heart of the **hexView** package is the `readBin()` function and the core facilities for working with `"raw"` binary objects in R code (e.g., `rawToChar()`); thanks to the R-core member(s) who were responsible for developing those features.

I would also like to thank the anonymous reviewer for useful comments on early drafts of this article.

Bibliography

IEEE Standard 754 for Binary Floating-Point Arithmetic. IEEE computer society, 1985. 4

R Development Core Team. *R Internals*. R Foundation for Statistical Computing, Vienna, Austria, 2006. URL `http://www.R-project.org`. ISBN 3-900051-14-3. 5

R. Ramanathan. *INTRODUCTORY ECONOMETRICS WITH APPLICATIONS*. Harcourt College, 5 edition, 2002. ISBN 0-03-034342-9. 6

Wikipedia. External data representation — wikipedia, the free encyclopedia, 2006a. URL `http://en.wikipedia.org/w/index.php?title=External_Data_Representation&oldid=91734878`. [Online; accessed 3-December-2006]. 4

Wikipedia. IEEE floating-point standard — wikipedia, the free encyclopedia, 2006b. URL `http://en.wikipedia.org/w/index.php?title=IEEE_floating-point_standard&oldid=89734307`. [Online; accessed 3-December-2006]. 4

Paul Murrell
Department of Statistics
The University of Auckland
New Zealand
`paul@stat.auckland.ac.nz`

FlexMix: An R Package for Finite Mixture Modelling

by Bettina Grün and Friedrich Leisch

Introduction

Finite mixture models are a popular method for modelling unobserved heterogeneity or for approximating general distribution functions. They are applied in a lot of different areas such as astronomy, biology, medicine or marketing. An overview on these models with many examples for applications is given in the recent monographs McLachlan and Peel (2000) and Frühwirth-Schnatter (2006).

Due to this popularity there exist many (standalone) software packages for finite mixture modelling (see McLachlan and Peel, 2000; Wedel and Kamakura, 2001). Furthermore, there are several different R packages for fitting finite mixture models available on CRAN. Packages which use the EM algo-