Visualising very long data vectors with the Hilbert curve
Description of the Bioconductor packages `HilbertVis` and `HilbertVisGUI`.

Simon Anders

European Bioinformatics Institute,
Hinxton, Cambridge, UK

sanders@fs.tum.de

2009-06-29

1 Introduction

In a number of different fields, intermediate results often take the form of extremely long data vectors with many millions of entries. In order to grasp the overall statistical structure and the distribution of features, specialdef visualisation techniques can be most helpful. The Bioconductor package "HilbertVis" offers such a tool.

In the present vignette, the package is demonstrated with random example data. To see the use of `HilbertVis` with real biological data (namely ChIP-Seq data), see the publication [And08b] or the vignette [And08a].

2 The problem

The command `makeRandomTestData` (which is provided only for demonstration purposes) produces a very long vector filled with random data

```
> library( HilbertVis )
> vec <- makeRandomTestData( )
```

This is a very long vector, with 10,000,000 entries:

```
> length(vec)
```

```
[1] 10000000
```

An obvious first way to inspect it is to make a needle plot it with a command like
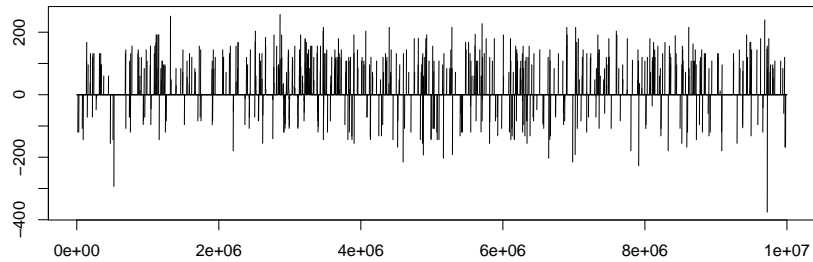
```
plot( vec, type="h" )
```

Figure 1: A simple plot of the example data vector, made with the function `plotLongVector`.
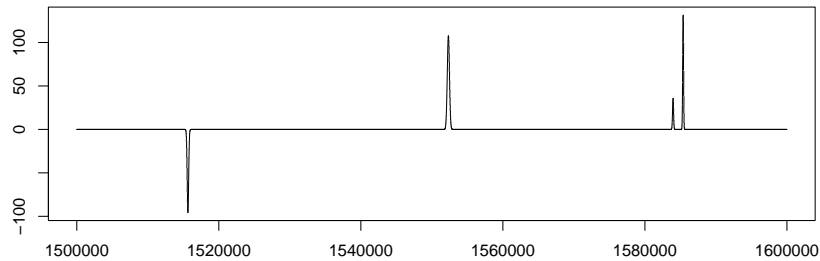

Figure 2: Zooming into a randomly chosen region of the data of Fig. 1.

This command takes very long to complete because so many needles are plotted on top of each other. The following command produces the same output in a more reasonable speed

```
> plotLongVector( vec )
```

From an inspection of this plot (Fig. 1), we may only conclude that the data vector contains a lot of peaks which seem to be distributed densely but randomly. But are all these peaks similar? Are they really evenly distributed? Are the individual needles in the plot single peaks, or several peaks which are plotted on top of each other?

The straight-forward solution is to zoom in and plot a region picked at random:

```
> start <- 1500000
> plotLongVector( vec[ start + 0:100000 ], offset=start )
```

In order to see whether this region (Fig. 2) is now typical, and whether all peaks look like these, we would have to make many more such plots.
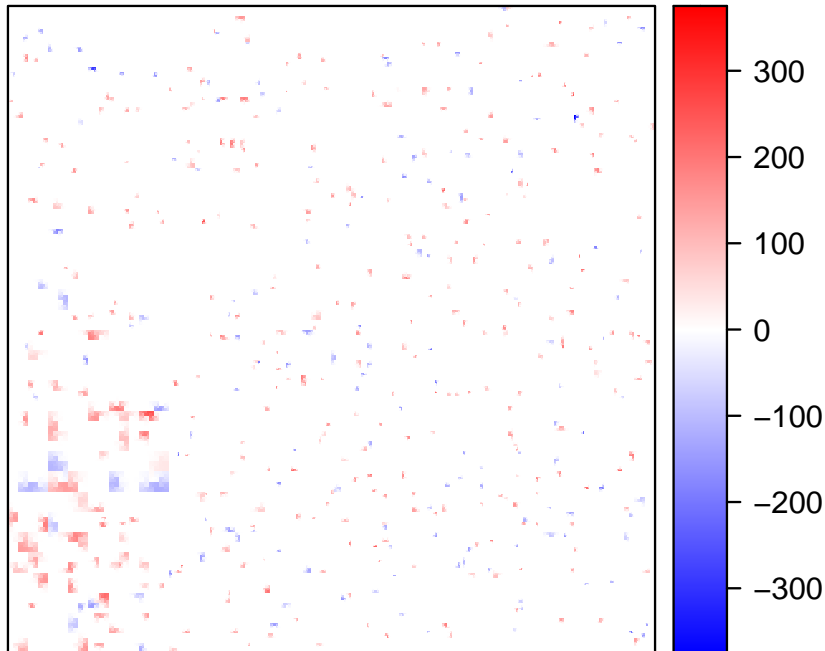
Figure 3: Hilbert curve plot of the example data vector from Fig. 1

3 Hilbert curves

The problem with Fig. 1 is that each pixel corresponds to several thousands of vector elements. Even if we made the figure much wider and print it finer, the horizontal axis can only hold a very limited amount of discernible values. The idea of the Hilbert curve visualisation is to use the space in both the horizontal and vertical direction in order to show more details.

The Hilbert curve plot is an approach to display an as detailed picture of the whole data vector as possible by letting each pixel of a large square represent a quite short part of the chromosome, coding with its colour for the maximum value in this short stretch, where the pixels are arranged such that neighbouring parts of the data vector appear next to each other in the square. Furthermore, parts which are not directly neighbouring but are in close distance should not be separated much in the square either. Fig. 3 shows our data vector visualised in this way.

In order to understand this plot you need to know how the pixels are arranged to fulfill the
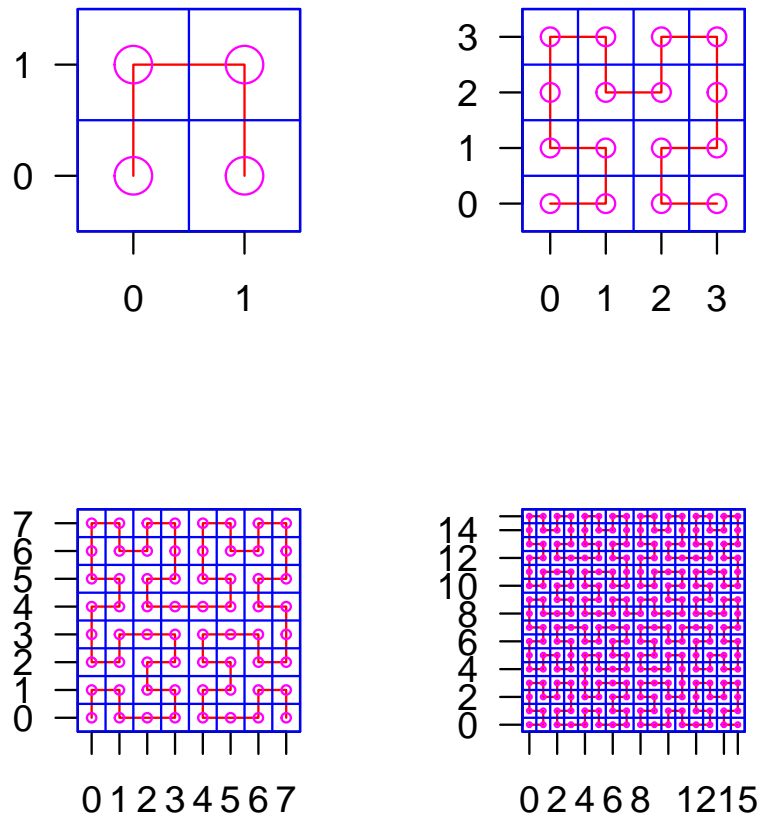
Figure 4: The first four levels of the Hilbert curve fractal.

requirements just outlined as well as possible. To my knowledge, the first to come up with the solution also used here was D. A. Keim in Ref. [Kei96] (where he used the technique to visualise long time-series data of stock-market prices). He went back to an old idea of Peano [Pea90] and Hilbert [Hil91], space-filling curves. Peano astonished the mathematics community at the end of the 19th century by presenting a continuous mapping of a line to a square, i.e., showed that a line can be folded up such that it passes through every point within a square, thus blurring the seemingly clear-cut distinction between one- and two-dimensional objects. Such a space-filling curve is a fractal, i.e., it has infinitely many corners and repeats its overall form in all levels of its details. Fig. 4 shows the first four level of the construction of Hilbert's variant of Peano's curve. Observe how at level k a line of length 2^{2k} passes through each "pixel" of a square of dimension $2^k \times 2^k$, and how this curve is produced connecting four copies (in different orientations) of the curve at the previous level, $k-1$.

For Fig. 3, a Hilbert curve at level 9 has been used, i.e., the image has $2^9 \times 2^9 = 262,144$ pixels. As `vec` has a length of 10 million elements, each pixel now represents around 38 elements. While we were unsure about how to interpret the needle-like peaks in Fig. 1, Fig. 3 gives a much clearer idea. Each of small blue or blue-red features is one peak, and its area is the width of the peak. (The colour gives the height.)

We can now see a feature of the data vector that was not noticeable in Figs. 1 and 2: While most of the peaks have an area of of one to ten pixels corresponding to peak width up to approx. 400), there is a region, in the lower left-hand corner, with much wider peaks. This spatial inhomogeneity is very obvious in the Hilbert plot but easy to miss with more conventional plots.

Before I show an example with biological data, I explain in the next section how the plots in this section were made.

4   Usage

Figure 4 has been produced with the function `plotHilbertCurve` which is provided just for demonstration purposes. Typically, it takes one parameter, the level of the curve. Here it is used within a loop to arrange the four plots in the figure.

```
> library( grid )
> pushViewport( viewport( layout=grid.layout( 2, 2 ) ) )
> for( i in 1:4 ) {
+     pushViewport( viewport(
+         layout.pos.row=1+(i-1)%/%2, layout.pos.col=1+(i-1)%%2 ) )
+     plotHilbertCurve( i, new.page=FALSE )
+     popViewport( )
+ }
```

To get the Hilbert plot of a data vector (Fig. 3), use the function `hilbertImage`:

```
> hMat <- hilbertImage( vec )
```

`hMat` is now a matrix of dimensions $2^9 \times 2^9$ (i.e., 512×512) that contains the intensity values for the plot. The function `hilbertImage` takes some optional arguments, for example `level`, in case you want another level (i.e. another matrix size) than the default 9.

To plot the matrix, we could just pass it to the standard functions `image` or `levelplot`. The function `showHilbertImage` is a convenient wrapper around `levelplot` that sets a suitable colour palette and supresses labels and tickmarks:

```
> showHilbertImage( hMat )
```

Hilbert curve data display

Displayed data
Previous | [1] me3.p10 | ▼ | Next

Bin under mouse cursor
Position: --- Value: ---

Full sequence
0 135,374,737

Displayed part of sequence
67,687,369 101,531,053

Effect of left mouse button
⊙ Zoom in 4× ○ Zoom in 64× ○ Linear plot

Zoom out
Zoom out 4× | Zoom out 64×

Pixel size
Coarser | Finer

Figure 5: The graphical user interface provided by `HilbertVisGUI`.

If you look at such an image on screen, you will notice that the `image` function does not display one matrix element as one pixel but rather scales the image according to the window size. This typically results in distracting artifacts, and hence, it is preferable to use the `display` function from the `EBImage` package that allows to suppress scaling. The 'mode' argument of `showHilbertImage` allows to use the `EBImage` package instead of the `lattice` package to display the matrix:

```
> showHilbertImage( hMat, mode="EBImage" )
```

5 Interactive tool for data exploration

Everything so far only required the package `HilbertVis`. Provided you have GTK+ and gtkmm installed on your system (see Appendix for more explanation) you can also install `HilbertVisGUI`

which provides a graphical tool to interactively explore Hilbert curve plots (Fig. 5).

To start the tool, load the package

```
> library( HilbertVisGUI )
```

and simply write

```
> hilbertDisplay( vec )
```

First notice the red pointer in the field labelled "Displayed part of sequence". As you move the mouse cursor through the Hilbert plot the pointer shows you where in the data vector ("sequence") you are. This visual feed-back helps to understand how the Hilbert curve "folds" the vector into the image. You can also read off the precise coordinate corresponding to the pixel under the mouse cursor (or rather: the midpoint of the range of coordinates ("bin") represented by this pixel) in the line above the gauge.

By clicking onto the image, you can zoom into the image. The program will select that quarter of the displayed curve that contains your mouse click and display it. The left gauge (labelled "full sequence") represents the full data vector, with the currently displayed part highlighted in red.

If you pass several vectors to `hilbertDisplay`, you can use the buttons labelled "Previous" and "Next" to switch back and forth between the data vectors in order to compare them.

By switching the "Effect of the left mouse button" to "Linear plot", you can get an ordinary 1D plot of the region around the pixel you clicked on. Click onto a peak and observe how you can inspect the peak's shape in the plot.

To do the linear plot, `hilbertDisplay` calls the R function `simpleLinPlot` defined as well in the `HilbertVisGUI` package. This is a simple wrapper around the function `plotLongVector` discussed earlier. You may want to have a look at its definition.

You can replace this function by supplying your own plotting function as the argument `plotFun` to `hilbertDisplay`. Your function must take two arguments that should be called `data` and `info`, as above, and will be filled in by `hilbertDisplay` with the displayed vector and information about where the user clicked and which part of the vector is being displayed. Try the following example to see the format of this data:

```
> dumpDataInsteadOfPlotting <- function( data, info ) {
+     str( data )
+     print( info )
+ }
> hilbertDisplay( vec, plotFunc=dumpDataInsteadOfPlotting )
```

Zoom in a bit, then switch to "linear plot" and click somewhere. `dumpDataInsteadOfPlotting` will be called and output such as the following appears on your R console:

```
num [1:10000000] 0 0 0 0 0 0 0 0 0 0 ...
$binLo
[1] 9555093

$bin
[1] 9555112

$binHi
[1] 9555131
```

```
$dispLo
[1] 7500001

$dispHi
[1] 10000001

$seqIdx
[1] 1

$seqName
[1] "vec"
```

The function `hilbertDisplay` has a number of useful optional arguments which are explained in its help page. Have a look there for details.

6 Three-channel Hilbert plots

Let us construct a second data vector that differs from `vec` only in a subtle way:

```
> vec2 <- vec
> vec2[ 7000000 : 8000000 ] <- vec[ 7010000 : 8010000 ]
```

If we compared the Hilbert plots of `vec` and `vec2` side-by-side we might miss the difference. Loading them both into the interactive tool, with

```
> hilbertDisplay( vec, vec2 )
```

makes spotting the difference easier: When clicking the "Previous" and "Next" buttons one notices that most peaks stay put while a few move a bit.

Another possibility is to overlay these two plots, displaying `vec` in red and `vec2` in green. The function `hilbertDisplayThreeChannels` start the interactive tool in this mode. Different from `hilbertDisplay`, this function expects the values for the channel to be normalised to values between 0 (corresponding to black) and 1 (corresponding to the channel's colour in full saturation). Hence, we divide both vectors by the maximum of `vec`.

```
> hilbertDisplayThreeChannel(
+    vec / max(vec), vec2 / max(vec), rep( 0, length(vec) ) )
```

The third argument is the vector for the blue channel. As we don't have a third vector, we simply pass a vector of the same length, filled with zeroes only.

In this display style, those peaks common to both vectors appear in yellow, while the those different are red and green (Fig. 6).

It might be desirable to have also a non-interactive version of this functionality. This can be done by calling `hilbertImage` for each data vector and then using the functionality of the `EBImage` package to overlay the three images in three colour channels. As the API of `EBImage` is due to be overhauled soon I do not describe this here as it would be outdated soon.

7 Reading in GFF and wiggle data

An important use for the `HilbertVis` package is genomic data. Imagine a vector the length of a human chromosome that gives for each base pair a score of some genome-position-dependent

Figure 6: A three-channel plot (only using the red and the green channel).

quantity. Such data is typically displayed as a "wiggle track" in genome browsers such as the UCSC browser at `http://genome.ucsc.edu/` [KSF+02].

The common formats to supply data for wiggle tracks are the General Feature Format (GFF) [DH+00], and the wiggle track format [Wig]. The function `makeWiggleVector` is supplied to read in such files and provide their content as a "wiggle vector", i.e., a vector with one element per base pair that gives the score value for this base pair. Such a wiggle vector can then be visualised in the same way as the data vector `vec` in the examples above. For details, see the help page for `makeWiggleVector`.

Another source of genomic data are aligned short reads from high-throughput sequencing experiments. See [And08a] for an explanation how to deal with such data.

8 Rle vectors

Long data vectors with are step-wise constant are more suitably dealt with in run-length encoding (RLE), i.e., instead of storing all values, one has two vectors, one that has the lengths of the constant stretches, one with the corresponding values. The `Rle` class in the `IRanges` package provides this functionality.

The functions `hilbertDisplay` and `hilbertDisplayThreeChannel` can both work with `Rle` vectors as well as with ordinary vectors (since version 1.3.1 of `HilbertVisGUI`).

A Installation

The packages `HilbertVis` and `HilbertVisGUI` are meant to be used with the statistics programming language R. While `HilbertVis` can be installed on any R system (with R version at least 2.7), you need to have `gtkmm` available on your system in order to install `HilbertVisGUI`. You need `HilbertVisGUI` only if you want to use the interactive tool described in Section . Don't worry, it is very easy to install gtkmm and the installer will tell you what to do.

There is also a stand-alone version of HilbertVis that can be used independent of R. See `http://www.ebi.ac.uk/~anders/hilbert` for more information.

A.1 Installing HilbertVis

To install `HilbertVis`, simply use the standard procedure explained on the Bioconductor web site, i.e., type in an R session

```
if (!requireNamespace("BiocManager", quietly=TRUE))
    install.packages("BiocManager")
BiocManager::install( "HilbertVis" )
```

A.2 Installing HilbertVisGUI

The GUI component uses the `gtkmm` library, the C++ bindings for the GTK+ widget system. [1]
Install `HilbertVisGUI` with `biocManager`, i.e., by typing

```
if (!requireNamespace("BiocManager", quietly=TRUE))
    install.packages("BiocManager")
BiocManager::install( "HilbertVisGUI" )
```

On platforms with binary packages (i.e., Linux and the like), the installer will check whether gtkmm is available. If it cannot find it, it will display an explanation how to install it.

On platforms with binary packages (Mac and Windows), this explanation appears once you load the package for the first time. You will be asked to simply download an automatic installer from a given web address.

B Session Info

```
> sessionInfo()

R version 4.5.0 RC (2025-04-04 r88126)
Platform: x86_64-pc-linux-gnu
Running under: Ubuntu 24.04.2 LTS

Matrix products: default
BLAS:   /home/biocbuild/bbs-3.21-bioc/R/lib/libRblas.so
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0  LAPACK version 3.12.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
```

---

[1] See `http://www.gtk.org/` for more information on the GTK+ project.

```
 [3] LC_TIME=en_GB             LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

time zone: America/New_York
tzcode source: system (glibc)

attached base packages:
[1] grid      stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
[1] HilbertVis_1.66.1 lattice_0.22-7

loaded via a namespace (and not attached):
[1] compiler_4.5.0 tools_4.5.0
```

C   References

References

[And08a] S. Anders. *Processing and visualisation of high-throughput sequencing with ShortRead and HilbertVis.* Vignette, to be distributed with Bioconductor package "ShortRead" (2008).

[And08b] S. Anders. *Visualisation of genomic data with space-filling curves.* Submitted (2008).

[DH⁺00] R. Durbin, D. Haussler, et al. *GFF (General Feature Format) specifications document.* http://www.sanger.ac.uk/Software/formats/GFF/GFF_Spec.shtml (2000).

[Hil91] D. Hilbert. *Über stetige Abbildungen einer Linie auf ein Flächenstück.* Mathematische Annalen **38** (1891), 459.

[Kei96] D. A. Keim. *Pixel-oriented visualization techniques for exploring very large data bases.* J. Comp. Graph. Stat. **5** (1996), 58.

[KSF⁺02] W. Kent, C. W. Sugnet, T. S. Furey, K. Roskin, T. H. Pringle, A. M. Zahler, D. Haussler. *The human genome browser at UCSC.* Genome Res. **12** (2002), 996.

[Pea90] G. Peano. *Sur une courbe qui remplit toute une aire plaine.* Mathematische Annalen **36** (1890), 157.

[Wig] *Wiggle track format (WIG).* http://genome.ucsc.edu/goldenPath/help/wiggle.html.