

The xmapcore Cookbook; Examples and Patterns for xmapcore

Tim Yates, Chris Wirth & Crispin Miller

August 25, 2010

Contents

1	Introduction	2
2	Initial Steps	2
2.1	Preamble	2
2.2	Loading the xmapcore package and connecting to the database	2
2.3	Disconnecting from the database	2
2.4	Finding your way around	2
3	Starting to fetch data	3
3.1	Getting it all	3
3.2	It's all in the details	4
3.3	The order of things	4
3.4	One thing to another	4
3.5	Finding a gene by its symbol	4
3.6	Going from a gene to its transcripts	5
3.7	The IN1 field in 'to' RangedData results	7
3.8	Microrray probeset mappings	8
3.8.1	Some subtleties with probeset mappings	9
4	Home, home on the Range	10
4.1	Finding things by their location	10
4.2	Combining range queries using xmap.range.apply	10
4.3	Further adventures with xmap.range.apply	12
5	Filtering probes	13
5.1	Exonic, intronic, intergenic and unreliable	13
5.2	The filtering methods	13
5.3	UTR filtering	15
5.4	Coding regions and UTRs	16
5.5	xmapcore's local cache	17
6	The path to the answer matters	18
6.1	The probeset boundary to commutativity	18
7	Utility methods	19
7.1	Plotting data	19
7.2	Customising plots	20

1 Introduction

xmapcore is a BioConductor package that provides annotation data and cross-mappings between, amongst other things, genes, transcripts, exons, proteins, domains and Affymetrix probesets. This document provides examples of its usage and recipes to help you get started.

2 Initial Steps

2.1 Preamble

xmapcore makes use of a MySQL database, X:Map, to provide its annotation data (downloadable from the X:Map website¹). A variety of different species are supported - each with their own separate database. If you've yet to install the package, please look at the installation instructions contained within INSTALL.pdf. These will walk you through the process.

2.2 Loading the xmapcore package and connecting to the database

Assuming everything is installed correctly, you should be able to load the package as normal:

```
> library(xmapcore)
```

You then need to tell xmapcore to connect to the annotation database:

```
> xmap.connect("hs-test")
```

```
Connected to xmapcore_homo_sapiens_58 (localhost)
Selected array 'HuEx-1_0' as a default.
```

A typical installation will support a variety of species, each with their own version of the X:Map database. You can specify which database to use in the function call. Here, for example, `hs-test` is the name of a database - in this case, one containing human data. If you enter `xmap.connect()` without any parameters, it will show you a list of possible databases to choose from².

2.3 Disconnecting from the database

To disconnect from the database, use:

```
> xmap.disconnect()
```

```
Disconnecting from xmapcore_homo_sapiens_58 (localhost)
```

Note that you do not have to disconnect before connecting to another database; xmapcore will do this automatically for you when `xmap.connect` is called.

2.4 Finding your way around

The majority of the calls in xmapcore fall into one of four categories: 'all' queries that fetch all known instances of an object type, 'details' queries that provide more detailed annotation for a list of IDs, 'to' queries that provide mappings between things (e.g. from genes to exons), and 'range' queries that find the things that lie between a pair of coordinates.

¹<http://xmap.picr.man.ac.uk/download>

²unless you only have one database defined, in which case it will simply connect you to that

3 Starting to fetch data

3.1 Getting it all

It is sometimes useful to get a list of all known objects of a certain type in the database. Available types are as follows (in alphabetical order): arrays, chromosomes, domains, est_exons, est_genes, est_transcripts, exons, genes, prediction_transcripts, probes, probesets, proteins, synonyms, and transcripts.

For example, to get a list of all the chromosomes in the human database, we can simply type:

```
> xmap.connect("hs-test")
```

```
Connected to xmapcore_homo_sapiens_58 (localhost)
```

```
Selected array 'HuEx-1_0' as a default.
```

```
> all.chromosomes()
```

```
[1] "11" "21" "7"  "Y"  "2"  "17" "22" "1"  "18" "13" "16" "6"  "X"  "3"  "9"
[16] "12" "14" "15" "20" "8"  "4"  "10" "19" "5"  "MT"
```

By passing `as.vector=FALSE` as a parameter, we can get more information than just the IDs:

```
> all.chromosomes(as.vector = FALSE)
```

	name	length
1	11	135006516
2	21	48129895
3	7	159138663
4	Y	59373566
5	2	243199373
6	17	81195210
7	22	51304566
8	1	249250621
9	18	78077248
10	13	115169878
11	16	90354753
12	6	171115067
13	X	155270560
14	3	198022430
15	9	141213431
16	12	133851895
17	14	107349540
18	15	102531392
19	20	63025520
20	8	146364022
21	4	191154276
22	10	135534747
23	19	59128983
24	5	180915260
25	MT	16569

You can get help on this family of functions by typing `?xmapcore.all` into an R session.

3.2 It's all in the details

Sometimes, you will have a list of IDs for which you require more detailed annotation. If this came from `xmapcore`, this can usually be achieved by passing the `as.vector=FALSE` parameter to the last function in your pipeline, but in cases where this is not possible, you can use the `details` methods.

For example, given a list of chromosome names, we can find their length as follows:

```
> chr.list = c("1", "2", "3")
> chromosome.details(chr.list)
```

```
  name  length
1     1 249250621
2     2 243199373
3     3 198022430
```

Or we can look for the details of a few microarray probes by their sequence:

```
> some.probes = probeset.to.probe("3855295")
> probe.details(some.probes)
```

```
          sequence probe_hit_count
1 GTCACCCTGCAACCGTGCCACGTGG          1
2 GTCATGTGGCGCCTGTTTCGACGCC          1
3 CAGGCTCTAGGACTGCGCGATGAGC          1
4 ACCAGGTCTGGCTCGCGGCGGACGT          1
```

It should be noted (and here seems as good a point as any) that Probe sequences inside the `xmapcore` database are not the exact physical sequences to be found on the corresponding array. Instead, we always return the sequence of the genomic target.

3.3 The order of things

When calling the following transformation queries (`X.to.Y`, and `X.details`), you should not rely on the order of the output being the same as the input order. Firstly, `RangedData` objects sort their results by the chromosome (space) they appear on, and secondly not all mappings are 1:1 (some are 1:many and some may be 1:0). If you need to look things up based on the input id you gave them, you should either subset based on the identifier column (ie: `d[d$stable_id == 'PTEN',]`), or you can split the results based on a column (see section 3.7)

3.4 One thing to another

In this section, we look at the `'to'` queries. The majority of mappings are available, and all take the form `X.to.Y`, as shown in Table 1. They are also described in the man pages (i.e. by typing `?xmapcore.to` when the `xmapcore` package is loaded).

3.5 Finding a gene by its symbol

For example, we can fetch a gene by its symbol, simply as follows:

```
> symbol.to.gene("TP53")
[1] "ENSG00000141510"
```

The function will also take a character vector. It should be noted that the input order of the parameters in the vector does not necessarily match the output order – objects will come back in the order that the database decides is optimal. If you want to match the results back to your input queries, you will need to use `as.vector=FALSE` and the `IN1` column (see section 3.7):

array	.to.	probeset
domain	.to.	gene, probeset, protein, transcript
probeset	.to.	cdnatranscript, domain, est_exon, est_gene est_transcript, exon, gene, hit, prediction_transcript probe, protein, transcript
probe	.to.	hit, probeset
gene	.to.	domain, exon, exon_probeset, probeset, protein symbol, synonym, transcript
transcript	.to.	cdnaprobeset, domain, exon, exon_probeset, gene, probeset protein, synonym
exon	.to.	gene, probeset, transcript
synonym	.to.	gene, transcript, est_gene, est_transcript
symbol	.to.	gene, transcript, est_gene, est_transcript
est_gene	.to.	est_exon, est_transcript, probeset
est_transcript	.to.	est_exon, est_gene, probeset
est_exon	.to.	est_gene, est_transcript, probeset
prediction_transcript	.to.	prediction_exon, probeset
protein	.to.	domain, gene, probeset, transcript

Table 1: Available mappings for X.to.Y in xmapcore

```
> symbol.to.gene(c("TP53", "SHH"))
[1] "ENSG00000141510" "ENSG00000164690"
```

Almost all of the functions in xmapcore take the parameter `as.vector`. By default, the commands return a character vector containing object IDs, but by setting this to `FALSE`, we can get a `RangedData`³ object containing our results.

```
> symbol.to.gene(c("TP53", "SHH"), as.vector = FALSE)
RangedData with 2 rows and 9 value columns across 2 spaces
      space      ranges |      IN1      stable_id      strand
<character> <IRanges> | <character> <character> <integer>
1      17 [ 7565257, 7590856] |      TP53 ENSG00000141510      -1
2       7 [155592680, 155604967] |      SHH ENSG00000164690      -1
      biotype      status
<character> <character>
1 protein_coding      KNOWN
2 protein_coding      KNOWN
                                     description
                                     <character>
1      tumor protein p53 [Source:HGNC Symbol;Acc:11998]
2 sonic hedgehog homolog (Drosophila) [Source:HGNC Symbol;Acc:10848]
      db_display_name      symbol      symbol_description
<character> <character> <character>
1 HGNC (curated)      TP53      tumor protein p53
2 HGNC (curated)      SHH sonic hedgehog homolog (Drosophila)
```

3.6 Going from a gene to its transcripts

```
> gene = symbol.to.gene(c("TP53", "SHH"))
> gene.to.transcript(gene)
```

³Part of the IRanges package. <http://www.bioconductor.org/packages/release/bioc/html/IRanges.html>

```

[1] "ENST00000269305" "ENST00000504290" "ENST00000510385" "ENST00000504937"
[5] "ENST00000455263" "ENST00000420246" "ENST00000445888" "ENST00000509690"
[9] "ENST00000514944" "ENST00000505014" "ENST00000508793" "ENST00000503591"
[13] "ENST00000413465" "ENST00000396473" "ENST00000419024" "ENST00000359597"
[17] "ENST00000414315" "ENST00000297261" "ENST00000441114" "ENST00000430104"
[21] "ENST00000435425" "ENST00000472308"

```

Again, we can pass `as.vector=FALSE` to get back a `RangedData` object:

```

> transcripts = gene.to.transcript(gene, as.vector = FALSE)
> transcripts

```

RangedData with 22 rows and 13 value columns across 2 spaces

	space	ranges	IN1	stable_id	strand
	<character>	<IRanges>	<character>	<character>	<integer>
1	17	[7571720, 7590856]	ENSG00000141510	ENST00000269305	-1
2	17	[7571720, 7578811]	ENSG00000141510	ENST00000504290	-1
3	17	[7571720, 7578811]	ENSG00000141510	ENST00000510385	-1
4	17	[7571720, 7578811]	ENSG00000141510	ENST00000504937	-1
5	17	[7571722, 7590799]	ENSG00000141510	ENST00000455263	-1
6	17	[7571722, 7590799]	ENSG00000141510	ENST00000420246	-1
7	17	[7571739, 7590805]	ENSG00000141510	ENST00000445888	-1
8	17	[7576853, 7590805]	ENSG00000141510	ENST00000509690	-1
9	17	[7577535, 7590745]	ENSG00000141510	ENST00000514944	-1
10	17	[7577844, 7590805]	ENSG00000141510	ENST00000505014	-1

	biotype	status	description	db_display_name	symbol
	<character>	<character>	<character>	<character>	<character>
1	protein_coding	KNOWN	NA	HGNC (curated)	TP53-001
2	retained_intron	KNOWN	NA	HGNC (curated)	TP53-006
3	retained_intron	KNOWN	NA	HGNC (curated)	TP53-007
4	retained_intron	KNOWN	NA	HGNC (curated)	TP53-008
5	nonsense_mediated_decay	KNOWN	NA	HGNC (curated)	TP53-004
6	nonsense_mediated_decay	KNOWN	NA	HGNC (curated)	TP53-005
7	protein_coding	KNOWN	NA	HGNC (curated)	TP53-002
8	protein_coding	KNOWN	NA	HGNC (curated)	TP53-015
9	protein_coding	KNOWN	NA	HGNC (curated)	TP53-013
10	retained_intron	KNOWN	NA	HGNC (curated)	TP53-009

	symbol_description	translation_start	translation_start_exon	translation_end
	<character>	<integer>	<integer>	<integer>
1		29	1057196	82
2		NA	NA	NA
3		NA	NA	NA
4		NA	NA	NA
5		29	1057196	48
6		29	1057196	33
7		26	1057220	82
8		22	1057222	74
9		29	1057196	74
10		NA	NA	NA

	translation_end_exon
	<integer>
1	1057205
2	NA
3	NA
4	NA

```

5           1057216
6           1057218
7           1057221
8           1057223
9           1057225
10          NA
...
<12 more rows>

```

All of the methods that take a vector of IDs will also take a RangedData object - so this will work:

```

> gene = symbol.to.gene("TP53", as.vector = FALSE)
> class(gene)

[1] "RangedData"
attr(,"package")
[1] "IRanges"

> gene.to.transcript(gene)

[1] "ENST00000269305" "ENST00000504290" "ENST00000510385" "ENST00000504937"
[5] "ENST00000455263" "ENST00000420246" "ENST00000445888" "ENST00000509690"
[9] "ENST00000514944" "ENST00000505014" "ENST00000508793" "ENST00000503591"
[13] "ENST00000413465" "ENST00000396473" "ENST00000419024" "ENST00000359597"
[17] "ENST00000414315"

```

3.7 The IN1 field in 'to' RangedData results

As you can see from the RangedData results of 'to' queries, each row contains a field called IN1. This corresponds to the initial query that led to the row being generated. So you can see for our last RangedData result, if we extract the IN1 column, we get the original 2 genes that we sent to gene.to.transcript:

```

> transcripts[["IN1"]]

[1] "ENSG00000141510" "ENSG00000141510" "ENSG00000141510" "ENSG00000141510"
[5] "ENSG00000141510" "ENSG00000141510" "ENSG00000141510" "ENSG00000141510"
[9] "ENSG00000141510" "ENSG00000141510" "ENSG00000141510" "ENSG00000141510"
[13] "ENSG00000141510" "ENSG00000141510" "ENSG00000141510" "ENSG00000141510"
[17] "ENSG00000141510" "ENSG00000164690" "ENSG00000164690" "ENSG00000164690"
[21] "ENSG00000164690" "ENSG00000164690"

```

This can be very useful when we wish to use, for example, merge, to combine a table of expression data with the annotation data supplied by xmapcore.

It can also be used with the split method to get the returned data into a list, i.e.:

```

> probesets = exon.to.probeset(gene.to.exon(symbol.to.gene("SHH")),
+   as.vector = F)
> split(probesets[["stable_id"]], probesets[["IN1"]])

$ENSE00001086614
[1] "3081229" "3081230" "3081231"

$ENSE00001086617
[1] "3081227"

```

```
$ENSE00001149618
[1] "3081225" "3081218" "3081223" "3081220" "3081224" "3081226" "3081221"
[8] "3081219" "3081222"
```

```
$ENSE00001676036
[1] "3081218"
```

```
$ENSE00001758320
[1] "3081218"
```

```
$ENSE00001800592
[1] "3081218"
```

3.8 Microarray probeset mappings

xmapcore also provides probe mappings for Affymetrix Exon microarrays. When you connect to a database, xmapcore will pick an array by default (if a mapping exists for that species).

```
> xmap.connect("hs-test")
```

```
Disconnecting from xmapcore_homo_sapiens_58 (localhost)
Connected to xmapcore_homo_sapiens_58 (localhost)
Selected array 'HuEx-1_0' as a default.
```

To select your array of choice, you can use the `xmap.array.type` method⁴:

```
> xmap.array.type("HuEx-1_0")
```

As with `xmap.connect` calling this function with no parameters will present you with a list of available arrays to choose from.

Probesets are treated like any other feature, so:

```
> exon.to.probeset(gene.to.exon(symbol.to.gene("TP53")))

[1] "3743933" "3743925" "3743926" "3743922" "3743923" "3743924" "3743920"
[8] "3743919" "3743917" "3743918" "3743915" "3743913" "3743912" "3743908"
[15] "3261181" "3114414" "3743909" "2588903" "3284222" "2846057" "3678168"
[22] "3743936" "3743928"
```

will find all of the probesets that map to exons in the gene TP53.

Again, we can call `probeset.details` or pass `as.vector=FALSE` to a `X.to.probeset` mapping to get the full details on a given probeset. Here's the details of the first probeset from our last query:

```
> exon.to.probeset(gene.to.exon(symbol.to.gene("TP53")), as.vector = FALSE)[1,
+ ]
```

	IN1	stable_id	array_name	probe_count	hit_score	gene_score
1	ENSE00001146308	3743933	HuEx-1_0	4	1	0
		transcript_score	exon_score	est_gene_score	est_transcript_score	
1		0	0	2	2	
		est_exon_score	prediction_transcript_score	prediction_exon_score		
1		0	0	0	0	
		protein_score	domain_score			
1		0	0			

⁴At the time of writing, most species only have one Array associated with them (some like *S. pombe* have none), this is therefore a function you will probably never have to use

As you can see, when you view a probesets details, you get back a load of score information. These scores are always 0, 1 or 2;

0. One or more of the probes miss the item of interest
1. All of the probes hit the item of interest once (and only once)
2. One or more of the probes hit more than one item of interest

So a score of 2 for `gene_score` means that one or more of the probes hits more than one gene (if it also has a hit score of 1, then this means that there is a region where 2 genes are overlapping)

3.8.1 Some subtleties with probeset mappings

Probeset mappings in X:Map are done in two ways. Probes are not only mapped to the entire genome, and their match locations recorded, but they are also mapped directly to cDNA sequences in order to pick up those probes and probesets that fall on exon/exon boundaries.

`probeset.to.cdna` and `transcript.to.cdna` provide the mappings needed to retrieve these probesets.

4 Home, home on the Range

4.1 Finding things by their location

It is also possible to provide a set of genomic regions and find the features they contain.

For example:

```
> genes.in.range("7", 1e+06, 1060000, 1)
```

```
[1] "ENSG00000073067"
```

will find all of the genes on the forward strand of chromosome '7' that lie between 1000000 and 1060000 nucleotides. All range queries expect the same set of parameters (chromosome, start, end and strand). As before, by default range queries return a character vector of database identifiers. Setting `as.vector=FALSE` will return detailed annotation in a `RangedData` object:

```
> genes = genes.in.range("7", 1e+06, 1060000, 1, as.vector = FALSE)
```

```
> genes
```

```
RangedData with 1 row and 8 value columns across 1 space
```

space	ranges	stable_id	strand	biotype
<character>	<IRanges>	<character>	<integer>	<character>
1	7 [1022835, 1029276]	ENSG00000073067	1	protein_coding
status				
<character>				
1	KNOWN			
				description
				<character>
1	cytochrome P450, family 2, subfamily W, polypeptide 1 [Source:HGNC Symbol;Acc:20243]			
db_display_name	symbol			
<character>	<character>			
1	HGNC (curated)	CYP2W1		
		symbol_description		
		<character>		
1	cytochrome P450, family 2, subfamily W, polypeptide 1			

The equivalent (`Xs.in.range`) functions will provide mappings for transcripts, exons and genes (for 'normal' genes as well as gene predictions and ESTs - see e.g. `?est_genes.in.range` for more details), proteins, domains, probesets and probes.

4.2 Combining range queries using `xmap.range.apply`

A common task is to iterate down a `RangedData` object and perform a function per row. The utility method `xmap.range.apply` offers similar functionality for `RangedData` objects. For example, we can find all probes that target the 3' end of each gene in `genes`, as follows:

First, we alter the `IRange` of these objects so they are just covering our region of interest:

```
> start(genes) = end(genes) - 1000
```

Then we use `xmap.range.apply` to perform a `probes.in.range` query for each 'row' of the `RangedData` object, `genes`.

```
> xmap.range.apply(genes, probes.in.range)
```

```

[1] "TCTGTGTTGGGGAGCGCCTGGCCAG" "AGCTCTTCTGCTGTTTGCCGGCCT"
[3] "TCCTGCAGAGGTACCGCTGCTGCC" "CGCCGGGCTTTTACCATGAGGCCG"
[5] "GCCCGGGCTTTTACCATGAGGCCGA" "CCGGGCTTTTACCATGAGGCCGAGG"
[7] "GGGCTTTTACCATGAGGCCGAGGGC" "CCCACAGCTCGGACTGCTCTGGGAG"
[9] "GTCAGCAACTGCTTCCGGTTACACC" "TGCTTCCGGTTACACCCAGGACTAC"
[11] "CTGAAGCTGCACTCCCACCCACCTA" "CTGCAGGGAGACAACGGGTGGCTGC"
[13] "GAGACAACGGGTGGCTGCATCCAGC" "GACAACGGGTGGCTGCATCCAGCCA"
[15] "CTGCATCCAGCCAGAGACAGGCGCA" "TGGGTGTCCTCAGCGTGCGAGCCCT"
[17] "GGTGTCTCAGCGTGCGAGCCCTGC" "TGTCTCAGCGTGCGAGCCCTGCAC"
[19] "CTCAGCGTGCGAGCCCTGCACCCCC" "ACTCCATTCCCGCTCTGGAACACT"
[21] "TCCATTCCCGCTCTGGAACACTTC" "TGTGCTGAGGAGGAGTCCGGCTGCA"
[23] "AGCCACTGGGGCCATGCGTATGACT" "AGGCACTGGCGCCAGAGGCTTCCTT"
[25] "AGCCCCTGAAGACAAGCAGCACTGC" "AAATGGAACACTGACCCGGTGCGG"
[27] "TGGAAACACTGACCCGGTGCGGTGG"

```

As with other queries, we can use the `as.vector` parameter to extract more details:

```

> xmap.range.apply(genes, probes.in.range, as.vector = F)

RangedData with 27 rows and 3 value columns across 1 space
  space      ranges |      sequence probe_hit_count
  <character> <IRanges> | <character> <numeric>
1      7 [1028280, 1028304] | TCTGTGTTGGGGAGCGCCTGGCCAG      1
2      7 [1028310, 1028334] | AGCTCTTCTGCTGTTTGCCGGCCT      1
3      7 [1028334, 1028358] | TCCTGCAGAGGTACCGCTGCTGCC      1
4      7 [1028398, 1028422] | CGCCGGGCTTTTACCATGAGGCCG      1
5      7 [1028399, 1028423] | GCCCGGGCTTTTACCATGAGGCCGA      1
6      7 [1028401, 1028425] | CCGGGCTTTTACCATGAGGCCGAGG      1
7      7 [1028403, 1028427] | GGGCTTTTACCATGAGGCCGAGGGC      1
8      7 [1028534, 1028558] | CCCACAGCTCGGACTGCTCTGGGAG      1
9      7 [1028598, 1028622] | GTCAGCAACTGCTTCCGGTTACACC      1
10     7 [1028607, 1028631] | TGCTTCCGGTTACACCCAGGACTAC      1
  strand
  <integer>
1      1
2      1
3      1
4      1
5      1
6      1
7      1
8      1
9      1
10     1
...
<17 more rows>

```

Another example, would be to find `est_genes` which overlap known genes;

```

> xmap.range.apply(symbol.to.gene(c("lama3", "tp53", "shh"), as.vector = F),
+   est_genes.in.range)

[1] "ENSESTG00007282567" "ENSESTG00007282658" "ENSESTG00007261775"
[4] "ENSESTG00007261856" "ENSESTG00007261871" "ENSESTG00007261967"
[7] "ENSESTG00007262010" "ENSESTG00007262047" "ENSESTG00007262483"
[10] "ENSESTG00007263129"

```

4.3 Further adventures with `xmap.range.apply`

`xmap.range.apply` can also be used to apply custom functions across a `RangedData` object. To take a contrived example, lets write a function that takes a `RangedData` object, and returns a character vector of the form `chr:start_location` (to provide labels for a graph, say):

```
> contrived.function = function(chromosome, location) {  
+   paste(chromosome, ":", location, sep = "")  
+ }
```

In a moment, we're going to use `xmap.range.apply` to map this down a set of transcripts, which we'll retrieve now:

```
> transcripts = gene.to.transcript(symbol.to.gene(c("shh", "tp53")),  
+   as.vector = F)
```

Before we can do this, though, we need to tell the function which columns in the `RangedData` object to map onto 'chromosome' and 'location', and also to tell it what datatype each of these parameters should be.

This is done using two additional parameters in `xmap.range.apply`: `filter` and `coerce`. The first defines the column names in the `RangedData` object we are interested in, the second, the functions we need to coerce these to the expected types:

```
> contrived.filter = c(chromosome = "space", location = "start")  
> contrived.coerce = c(as.character, as.numeric)
```

The latter is needed because we can't always rely on R to correctly guess the data type we're expecting ('space' in a `RangedData` object will often be converted to a factor, not a character vector, for example). Once these are defined, we can then fire off our apply statement, as follows:

```
> xmap.range.apply(transcripts, contrived.function, contrived.filter,  
+   contrived.coerce)  
  
[1] "17:7571720" "17:7571720" "17:7571720" "17:7571720" "17:7571722"  
[6] "17:7571722" "17:7571739" "17:7576853" "17:7577535" "17:7577844"  
[11] "17:7578434" "17:7578547" "17:7565257" "17:7571740" "17:7576537"  
[16] "17:7576625" "17:7578138" "7:155592680" "7:155592733" "7:155592734"  
[21] "7:155592744" "7:155599276"
```

As you can see, it's pretty easy to bend `xmap.range.apply` to your will!

5 Filtering probes

5.1 Exonic, intronic, intergenic and unreliable

Probesets in xmapcore are defined as being either exonic, intronic, intergenic or unreliable. These terms are best described by:

- **Exonic:** Probesets are classed as exonic if all of their probes map to the genome only once, and every one of these mappings falls within an exon boundary.
- **Intronic:** Probesets are classed as intronic if all of their probes map to the genome only once, but at least one probe misses an exon region, but still falls within the boundary of a gene.
- **Intergenic:** Probesets are classed as intergenic if all of their probes map to the genome only once, but at least one probe misses all the known genes.
- **Unreliable:** Probesets with one or more multi-targetting probes, or with one or more probes that do not map to the genome, are classed as unreliable.

These categories are a very broad filter. An unreliable probeset may have 3 probes which all hit in a single location and all hit an exon, but the single probe which misses mapping to the genome will result in the entire probeset being classed as to unreliable.

5.2 The filtering methods

To filter a collection of probesets by their relevant category, there are eponymous methods for you to use. For example, let's look at the probesets that are hitting around the gene for TP53:

```
> g = symbol.to.gene("tp53")
> ps = gene.to.probeset(g)
> ps
```

```
[1] "3151499" "3743940" "3743927" "3743921" "3743922" "3571368" "3743920"
[8] "3743910" "3603397" "3743938" "3743939" "3743936" "3743914" "2624339"
[15] "3030447" "3743929" "2412311" "2891220" "3016635" "3676707" "3743919"
[22] "2526753" "3854008" "3743928" "3743932" "3178489" "2670237" "3472536"
[29] "3743925" "3659940" "3960291" "3402795" "3743915" "2691631" "3675098"
[36] "3844041" "3362335" "3453474" "3743917" "2549141" "2923872" "3292431"
[43] "3551386" "3055766" "3286319" "2591241" "3553844" "3976408" "3743918"
[50] "3615876" "3246354" "3301451" "3555253" "3042658" "2410077" "2662284"
[57] "2746540" "2880800" "3462599" "3535951" "3638759" "3645143" "3655014"
[64] "3975448" "3743913" "3743916" "3743911" "2733426" "2752876" "3292430"
[71] "3725396" "2515345" "3275742" "3863023" "3743912" "3474440" "3743908"
[78] "3743930" "2830115" "3838637" "3743926" "2815874" "3311364" "3373304"
[85] "3072177" "3129448" "2638573" "3553618" "2406381" "2724073" "3578124"
[92] "3743905" "3743907" "2897724" "2907497" "3017565" "3338582" "3336154"
[99] "3261181" "3743931" "3781167" "2921371" "3666646" "3744997" "3743923"
[106] "2624089" "2363573" "3982418" "2867313" "3190732" "3426256" "3645871"
[113] "3743933" "3743935" "2457486" "2660118" "3674716" "2631438" "3728749"
[120] "3845706" "3834113" "3114414" "2903221" "2920908" "3800750" "3840377"
[127] "2666042" "2669390" "3227170" "3592447" "3721093" "3758826" "2974704"
[134] "3936282" "2528976" "3743924" "3743909" "2326133" "3462174" "2500292"
[141] "3257588" "2907507" "2588907" "3470332" "3841120" "2712842" "2646089"
[148] "2735060" "3585206" "3825799" "3961893" "3036651" "2374337" "2809843"
[155] "3824999" "3755769" "2350443" "2963365" "3831841" "4005824" "2494487"
[162] "2713240" "3248968" "3328152" "3471746" "3720274" "2622458" "3109991"
```

```

[169] "3178486" "3506008" "2685561" "3174530" "3706059" "3730986" "3626659"
[176] "3132417" "3456614" "2588903" "3755160" "2939930" "3497859" "3728036"
[183] "2328589" "3415635" "2521151" "4052397" "3549411" "3954787" "2340893"
[190] "2706519" "3768525" "3828181" "3936467" "2329359" "3067623" "3259813"
[197] "3869300" "3118532" "3809794" "2393742" "3619444" "3630186" "3770554"
[204] "3724339" "2931814" "3877306" "2335921" "2337699" "3083149" "3264433"
[211] "3284222" "3629932" "3975187" "2796100" "3924954" "3096572" "3920294"
[218] "3933310" "2757577" "3559795" "3740566" "2721787" "3939257" "4034920"
[225] "4035055" "2814193" "3615053" "2320372" "3341449" "2644485" "2325164"
[232] "2863213" "3454391" "3648904" "2846057" "3678168"

```

We can then see probesets that match each category:

```
> exonic(ps)
```

```
Building probeset specificity cache.....done
```

```

[1] "3743922" "3743920" "3743936" "3743919" "3743928" "3743925" "3743915"
[8] "3743917" "3743918" "3743913" "3743912" "3743908" "3743926" "3743923"
[15] "3743924" "3743909"

```

```
> intronic(ps)
```

```

[1] "3743940" "3743927" "3743921" "3743910" "3743938" "3743939" "3743914"
[8] "3743929" "3743932" "3743916" "3743911" "3743930" "3743905" "3743907"
[15] "3743931" "3743935"

```

```
> intergenic(ps)
```

```
[1] "3743933"
```

```
> unreliable(ps)
```

```

[1] "3151499" "3571368" "3603397" "2624339" "3030447" "2412311" "2891220"
[8] "3016635" "3676707" "2526753" "3854008" "3178489" "2670237" "3472536"
[15] "3659940" "3960291" "3402795" "2691631" "3675098" "3844041" "3362335"
[22] "3453474" "2549141" "2923872" "3292431" "3551386" "3055766" "3286319"
[29] "2591241" "3553844" "3976408" "3615876" "3246354" "3301451" "3555253"
[36] "3042658" "2410077" "2662284" "2746540" "2880800" "3462599" "3535951"
[43] "3638759" "3645143" "3655014" "3975448" "2733426" "2752876" "3292430"
[50] "3725396" "2515345" "3275742" "3863023" "3474440" "2830115" "3838637"
[57] "2815874" "3311364" "3373304" "3072177" "3129448" "2638573" "3553618"
[64] "2406381" "2724073" "3578124" "2897724" "2907497" "3017565" "3338582"
[71] "3336154" "3261181" "3781167" "2921371" "3666646" "3744997" "2624089"
[78] "2363573" "3982418" "2867313" "3190732" "3426256" "3645871" "2457486"
[85] "2660118" "3674716" "2631438" "3728749" "3845706" "3834113" "3114414"
[92] "2903221" "2920908" "3800750" "3840377" "2666042" "2669390" "3227170"
[99] "3592447" "3721093" "3758826" "2974704" "3936282" "2528976" "2326133"
[106] "3462174" "2500292" "3257588" "2907507" "2588907" "3470332" "3841120"
[113] "2712842" "2646089" "2735060" "3585206" "3825799" "3961893" "3036651"
[120] "2374337" "2809843" "3824999" "3755769" "2350443" "2963365" "3831841"
[127] "4005824" "2494487" "2713240" "3248968" "3328152" "3471746" "3720274"
[134] "2622458" "3109991" "3178486" "3506008" "2685561" "3174530" "3706059"
[141] "3730986" "3626659" "3132417" "3456614" "2588903" "3755160" "2939930"
[148] "3497859" "3728036" "2328589" "3415635" "2521151" "4052397" "3549411"
[155] "3954787" "2340893" "2706519" "3768525" "3828181" "3936467" "2329359"
[162] "3067623" "3259813" "3869300" "3118532" "3809794" "2393742" "3619444"

```

```
[169] "3630186" "3770554" "3724339" "2931814" "3877306" "2335921" "2337699"
[176] "3083149" "3264433" "3284222" "3629932" "3975187" "2796100" "3924954"
[183] "3096572" "3920294" "3933310" "2757577" "3559795" "3740566" "2721787"
[190] "3939257" "4034920" "4035055" "2814193" "3615053" "2320372" "3341449"
[197] "2644485" "2325164" "2863213" "3454391" "3648904" "2846057" "3678168"
```

Note that since only a single probe in a probeset needs to be intergenic for the entire probeset to be categorised as such, it is possible for a probeset that mostly maps to a gene to be flagged as intergenic.

All of these functions, will take an `exclude` parameter that results in an inverted list. So to get all of the probesets that are not unreliable, we can do:

```
> unreliable(ps, exclude = T)

[1] "3743940" "3743927" "3743921" "3743922" "3743920" "3743910" "3743938"
[8] "3743939" "3743936" "3743914" "3743929" "3743919" "3743928" "3743932"
[15] "3743925" "3743915" "3743917" "3743918" "3743913" "3743916" "3743911"
[22] "3743912" "3743908" "3743930" "3743926" "3743905" "3743907" "3743931"
[29] "3743923" "3743933" "3743935" "3743924" "3743909"
```

Since anything that is not unreliable must hit the genome somewhere the following should evaluate to TRUE:

```
> sort(unreliable(ps, exclude = T)) == sort(c(intronic(ps), exonic(ps),
+      intergenic(ps)))

[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[16] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[31] TRUE TRUE TRUE
```

5.3 UTR filtering

The function `utr.probesets` can be used to find those probesets that hit UTRs:

```
> gene = symbol.to.gene("tp53")
> probesets = gene.to.probeset(gene)
> utr.probesets(probesets)

[1] "3743926" "3743927" "3743928" "3743929" "3743930" "3743931" "3743932"
[8] "3743933" "3743938" "3743939" "3743940" "3743925" "3743921" "3743922"
[15] "3743923" "3743924" "3743908" "3743909" "3743910" "3743911" "3743912"
[22] "3743935"
```

In this form, the function will attempt to match probesets to all possible transcripts and then filter by UTR, so any UTR targeting probeset will be found.

You can also pass a vector of transcripts in, in which case the search is limited to these:

```
> transcripts = gene.to.transcript(gene)[1:2]
> transcripts

[1] "ENST00000269305" "ENST00000504290"

> utr.probesets(probesets, transcripts)

[1] "3743926" "3743927" "3743928" "3743929" "3743930" "3743931" "3743932"
[8] "3743933" "3743938" "3743939" "3743940" "3743908"
```

So these probesets have at least one probe in the UTR regions of the first two transcripts of TP53. It is also possible to limit the search to only the 3' or 5' UTR:

```
> utr.probesets(probesets, transcripts, end = "3")
```

```
[1] "3743908"
```

```
> utr.probesets(probesets, transcripts, end = "5")
```

```
[1] "3743926" "3743927" "3743928" "3743929" "3743930" "3743931" "3743932"  
[8] "3743933" "3743938" "3743939" "3743940"
```

It is also possible to omit the list of probesets and the `utr.probeset` function will generate one for you:

```
> utr.probesets(NULL, transcripts)
```

```
[1] "3743926" "3743927" "3743928" "3743929" "3743930" "3743931" "3743932"  
[8] "3743933" "3743938" "3743939" "3743940" "3743908"
```

Or you can get a list of probesets which fall inside the coding region of a transcript by using the complementary `coding.probesets` function:

```
> coding.probesets(NULL, transcripts)
```

```
[1] "3743909" "3743910" "3743911" "3743912" "3743913" "3743914" "3743915"  
[8] "3743916" "3743917" "3743918" "3743919" "3743920" "3743921" "3743922"  
[15] "3743923" "3743924" "3743925" "3743935" "3743936" "3743908"
```

5.4 Coding regions and UTRs

It is possible to manipulate a list of transcripts so that you find the genomic location of their coding region, or UTR. This is done using the two complementary methods `transcript.to.utrrange` and `transcript.to.coding.range`. Here for example is the normal details for a given transcript `ENST00000417324` which falls on the reverse strand of Chromosome 1 in human:

```
> transcript.details("ENST00000417324")
```

```
RangedData with 1 row and 12 value columns across 1 space
```

	space	ranges	stable_id	strand	biotype
	<character>	<IRanges>	<character>	<integer>	<character>
1	1	[34554, 36081]	ENST00000417324	-1	protein_coding
	status	description	db_display_name	symbol	symbol_description
	<character>	<character>	<character>	<character>	<character>
1	KNOWN	NA	HGNC (curated)	FAM138A-001	
	translation_start	translation_start_exon	translation_end	translation_end_exon	
	<integer>	<integer>	<integer>	<integer>	
1	346	1329474	37	1329476	

We can also get the genomic coordinates of its coding region:

```
> transcript.to.coding.range("ENST00000417324")
```

```
RangedData with 1 row and 12 value columns across 1 space
```

	space	ranges	stable_id	strand	biotype
	<character>	<IRanges>	<character>	<integer>	<character>
1	1	[35138, 35734]	ENST00000417324	-1	protein_coding
	status	description	db_display_name	symbol	symbol_description
	<character>	<character>	<character>	<character>	<character>
1	KNOWN	NA	HGNC (curated)	FAM138A-001	
	translation_start	translation_start_exon	translation_end	translation_end_exon	
	<integer>	<integer>	<integer>	<integer>	
1	346	1329474	37	1329476	

Or, we can get the regions covered by the UTR of this same transcript:

```
> transcript.to.utr.range("ENST00000417324")
```

```
RangedData with 2 rows and 3 value columns across 1 space
  space      ranges |      IN1      strand      prime
<character> <IRanges> | <character> <numeric> <factor>
1         1 [35735, 36081] | ENST00000417324      -1         5
2         1 [34554, 35137] | ENST00000417324      -1         3
```

Again, these two methods take an optional 'end' parameter to specify which end of the transcript you want to work with:

```
> transcript.to.coding.range("ENST00000417324", end = "3")
```

```
RangedData with 1 row and 12 value columns across 1 space
  space      ranges |      stable_id      strand      biotype
<character> <IRanges> | <character> <integer> <character>
1         1 [35138, 36081] | ENST00000417324      -1 protein_coding
  status description db_display_name      symbol symbol_description
<character> <character> <character> <character> <character>
1      KNOWN      NA      HGNC (curated) FAM138A-001
  translation_start translation_start_exon translation_end translation_end_exon
      <integer>      <integer>      <integer>      <integer>
1           346           1329474           37           1329476
```

```
> transcript.to.utr.range("ENST00000417324", end = "3")
```

```
RangedData with 1 row and 3 value columns across 1 space
  space      ranges |      IN1      strand      prime
<character> <IRanges> | <character> <numeric> <factor>
1         1 [34554, 35137] | ENST00000417324      -1         3
```

5.5 xmapcore's local cache

You may have noticed that the first time we called `exonic` on page 14, `xmapcore` told us that it was 'Building probeset specificity cache.....'. This cache allows us to filter large numbers of probesets much quicker than doing each in turn. However, for smaller queries of less than about 1000 probesets (on our system) it is quicker not to suffer the 1s load time for the cache to be loaded into memory from disk. If you are running lots of these filtering queries on small numbers of probesets (in a loop or apply, for example), it might be worth turning the cache off whilst you run them:

```
> xmap.toggle.caching()
```

```
[1] FALSE
```

Now, the cache (on by default) has been turned off. (This can be seen by the return value of `FALSE`). Calling `toggle` again will turn it back on:

```
> xmap.toggle.caching()
```

```
[1] TRUE
```

Currently we cache the probeset specificity data used for the filters, described above, and the calls to `all.xxx`. The cache is stored in `.xmapcore/cache`.

6 The path to the answer matters

6.1 The probeset boundary to commutativity

In general, `xmapcore` is commutative. For example, if you go from a list of genes to transcripts and back again, you will end up where you started:

```
> genes = symbol.to.gene(c("pten", "shh"))
> genes

[1] "ENSG00000171862" "ENSG00000164690"

> transcripts = gene.to.transcript(genes)
> transcripts

[1] "ENST00000371953" "ENST00000487939" "ENST00000462694" "ENST00000498703"
[5] "ENST00000472832" "ENST00000297261" "ENST00000441114" "ENST00000430104"
[9] "ENST00000435425" "ENST00000472308"

> genes = transcript.to.gene(transcripts)
> genes

[1] "ENSG00000171862" "ENSG00000164690"

> gene.to.symbol(genes)

[1] "PTEN" "SHH"
```

However, with probesets, this commutativity is broken. This is because they (even exonic probesets) can hit genes, transcripts or exons other than those in the list that you passed in – for example:

```
> genes = symbol.to.gene(c("pten", "shh"))
> genes

[1] "ENSG00000171862" "ENSG00000164690"

> probesets = exonic(gene.to.probeset(genes))
> probesets

[1] "3256701" "3256753" "3256751" "3256781" "3256754" "3256783" "3256703"
[8] "3256740" "3256752" "3256706" "3256726" "3256782" "3256738" "3256702"
[15] "3256741" "3256716" "3256784" "3256755" "3256756" "3081227" "3081225"
[22] "3081218" "3081229" "3081223" "3081230" "3081220" "3081224" "3081226"
[29] "3081221" "3081219" "3081222"

> genes = probeset.to.gene(probesets)
> genes

[1] "ENSG00000171862" "ENSG00000213613" "ENSG00000164690"

> gene.to.symbol(genes)

[1] "PTEN"          "RP11-380G5.3" "SHH"
```

Indeed, any mapping involving hits, probes or probesets, are not commutative.

7 Utility methods

7.1 Plotting data

Currently, the `xmap.gene.plot` method can plot 4 different styles of graph. The first 'real', is the gene drawn as it would be seen in (for example) X:Map, with exons, introns, and absolute locations (Figure 1);

```
> xmap.gene.plot(symbol.to.gene("tp53"), style = "real")
```

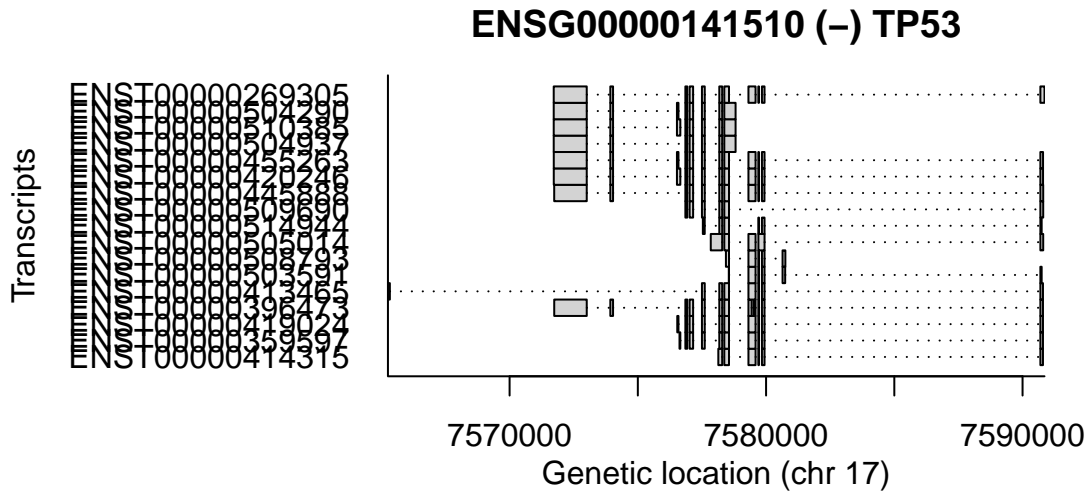


Figure 1: An 'real' gene plot

The 'injoined' style aligns the transcripts up to the left hand side of the graph (Figure 2)

```
> xmap.gene.plot(symbol.to.gene("tp53"), style = "injoined")
```

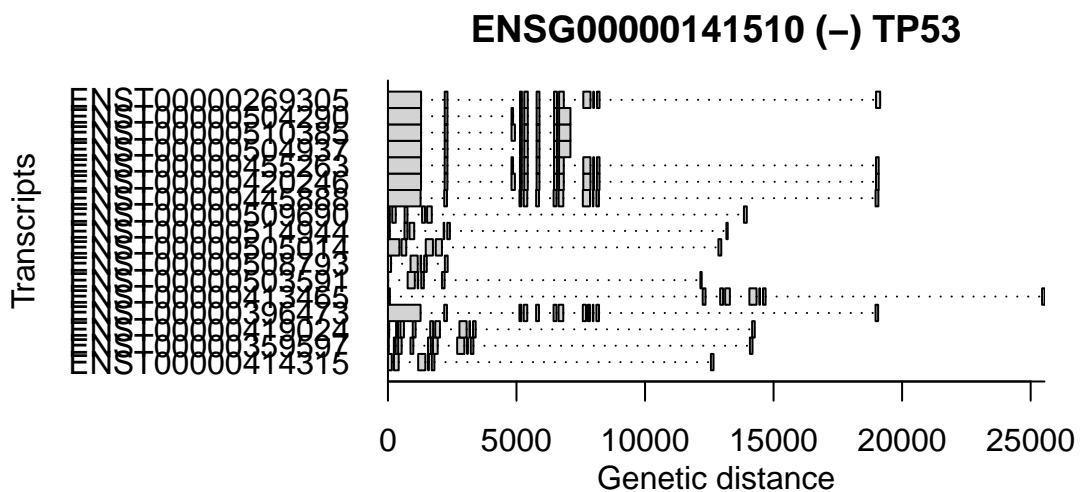


Figure 2: An 'injoined' gene plot

'jointed' plots only the exons with no intronic regions (Figure 3) (essentially, the cDNA sequence with the exon/exon boundaries shown).

```
> xmap.gene.plot(symbol.to.gene("tp53"), style = "jointed")
```

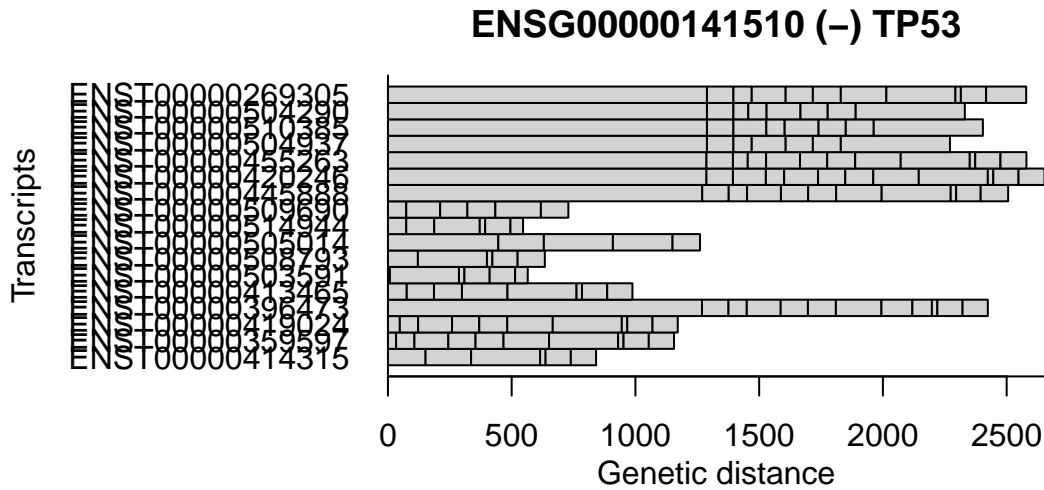


Figure 3: An 'jointed' gene plot

Finally, 'stacked' shows all exons the same size and stacked in order (Figure 4)

```
> xmap.gene.plot(symbol.to.gene("tp53"), style = "stacked")
```

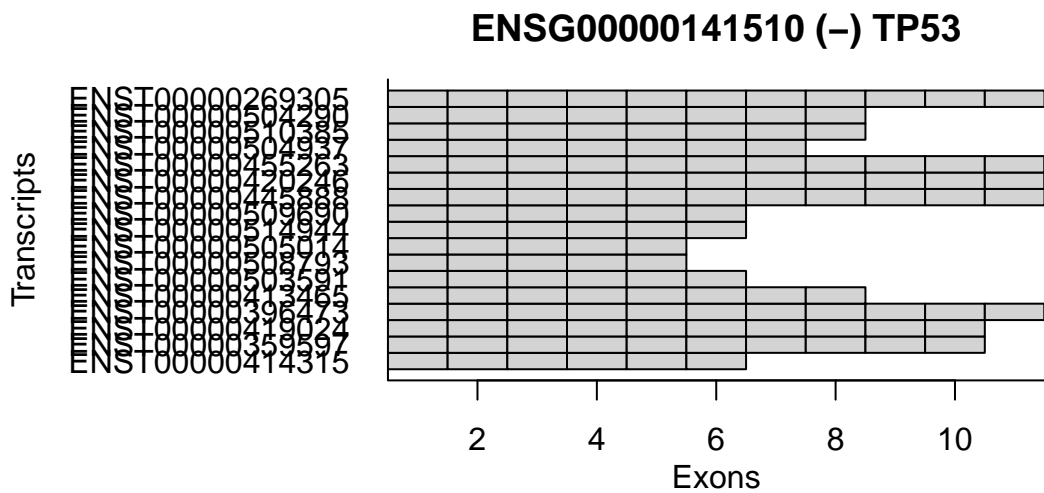


Figure 4: An 'stacked' gene plot

7.2 Customising plots

The `xmap.gene.plot` method takes a parameter `rect.func` which is called each time a transcript is ready for plotting. This function basically takes one parameter, which is a `data.frame` containing

columns gene, transcript, exon, rect.x1, rect.y1, rect.x2 and rect.y2. It should therefore be possible to render exons in a different way, eg to show expression data readings for them. This example shows you how to render a jointed graph (Figure 5), so that each transcript is in a random colour (obviously, we could render each exon in turn instead);

```
> randomcolor = function() {
+   rgb(sample(0:255, 1)/255, sample(0:255, 1)/255, sample(0:255,
+   1)/255)
+ }
> mycolourfunc = function(d) {
+   rect(d[, "rect.x1"], d[, "rect.y1"], d[, "rect.x2"], d[,
+   "rect.y2"], col = randomcolor())
+ }

> xmap.gene.plot(symbol.to.gene("tp53"), style = "jointed", rect.func = mycolourfunc)
```

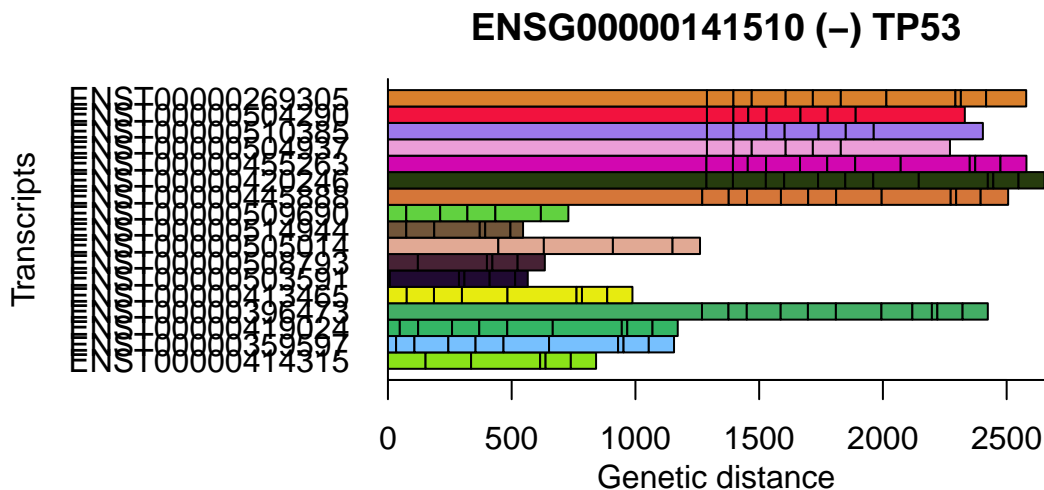


Figure 5: A custom 'jointed' gene plot

In the data.frame, which is passed to xmap.gene.plot, each row represents a single exon. As such, if we wish to render each exon individually, we can do so by performing an apply function over each row in the data.frame, as shown in Figure 6:

```
> mycolourfunc = function(d) {
+   apply(d, 1, function(row) {
+     rect(row["rect.x1"], row["rect.y1"], row["rect.x2"],
+     row["rect.y2"], col = randomcolor())
+   })
+ }

> xmap.gene.plot(symbol.to.gene("tp53"), style = "jointed", rect.func = mycolourfunc)
```

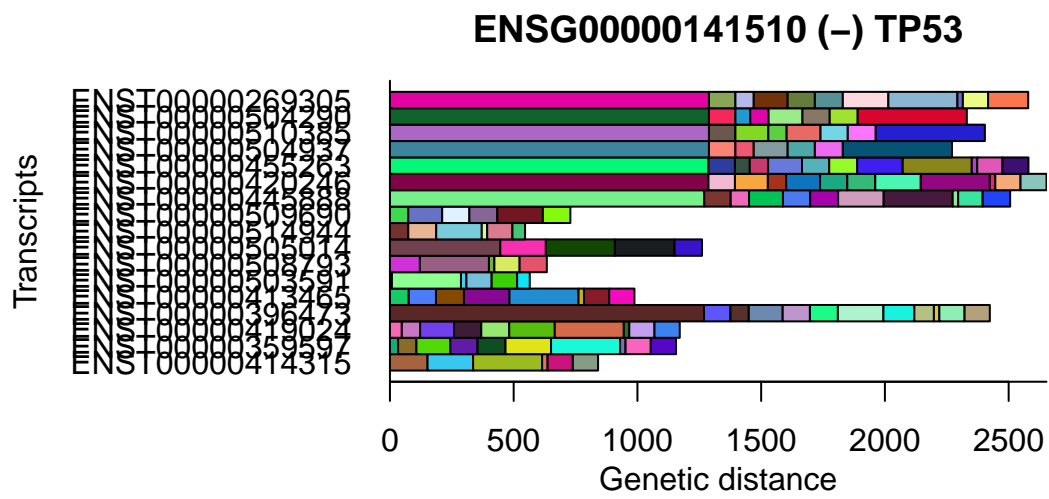


Figure 6: A custom 'jointed' gene plot with individually coloured exons