The annmap Cookbook; Examples and Patterns for annmap

Tim Yates, Chris Wirth & Crispin Miller

November 1, 2013

Contents

1 Introduction

annmap 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

annmap makes use of a MySQL database, annmap, to provide its annotation data (downloadable from the Annmap website[1]). A variety of different species are supported - each with their own separate database.

This document assumes that you have downloaded and installed one of the available MySQL databases (as described in the INSTALL vignette that came with this package).

It is also possible to connect to the annmap webservice instead of installing your own database – with some caveats. Please see the section 2.3 for details.

2.1 Preamble

This cookbook is designed to walk you through the annmap functions from first principles to you being able to generate plots such as Figure 1 for your Next-Generation Sequencing data.

```
> gene = symbolToGene( 'SHH' )
> strand( gene ) = '*'
> ngsBridgePlot( gene, bamfileData )
```

[1]http://annmap.cruk.manchester.ac.uk/download

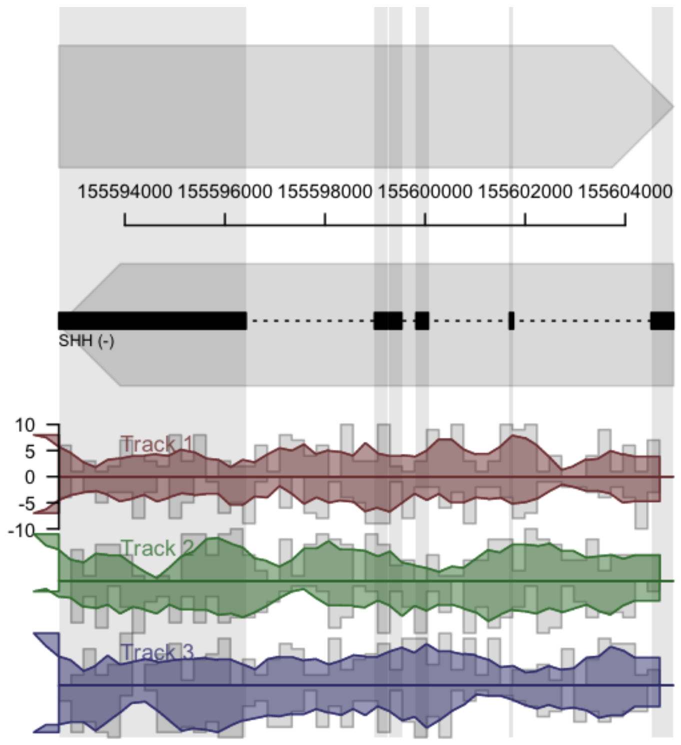

Figure 1: Dummy (in this example) NGS data shown alongside SHH. Grey boxes represent individual values. Coloured regions represent smoothed data.

2.2 Loading the `annmap` package and connecting to the database

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

```
> library( annmap )
```

If this is your first time running the package, you might need to create a datasource. These are stored by default in a folder called `.annmap` in your home directory, but this location can be customised by setting an environment variable `ANNMAP_HOME` to point to a folder of your choosing. To add a datasource, we will use the `annmapAddConnection` method.

Assuming you have installed the `annmap_homo_sapiens_66` database into a MySQL instance running on your local machine accessed by the username `'test'` and with a blank password, you can run:

```
> annmapAddConnection( 'human66', 'homo_sapiens', '66', username='test' )
```

You then need to tell `annmap` to connect to this annotation database:

```
annmapConnect( 'human66' )
```

```
Connected to annmap_homo_sapiens_66 (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 `annmap` database. You can specify which database to use in the function call. Here, for example, `human66` is the name we gave to our new connection. If you enter `annmapConnect()` without any parameters, it will show you a list of possible databases to choose from[2].

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

2.3 Using the annmap webservice

It's also possible to connect to the annmap webservice to get your data.

This is slower than using a locally installed database, less reliable as it relies on the annmap website which is occasionally taken down for updating.

Also, the annmap website operates a 'one in one out' rule, so when a new species version is added, the oldest one is removed to save disk space.

Those caveats aside, it's a great way of testing things out without the need to install MySQL and the database.

To connect to the webservice, simply call:

```
> annmapConnect( use.webservice=TRUE )
```

and you will be presented with a list of available connections to use.

As with the non-webservice connection, you can also do:

```
> annmapConnect( 'homo_sapiens.72', use.webservice=TRUE )
```

To connect to a known named webservice connection (so long as it exists).

2.4 Disconnecting from the database

To disconnect from the database, use:

```
annmapDisconnect()
```

```
Disconnecting from annmap_homo_sapiens_66 (localhost)
```

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

2.5 Overview

The majority of the calls in `annmap` 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. These are summarised in table 1.

		TO																
		Gene	Transcript	Exon	EST Gene	EST Transcript	EST Exon	Prediction Transcript	Prediction Exon	Probeset	Probe	Hit	Protein	Domain	Symbol		InRange	Details
From	Gene	N	Y	Y	N	N	N	N	N	Y	N	N	Y	Y	Y		Y	Y
	Transcript	Y	N	Y	N	N	N	N	N	Y	N	N	Y	Y	N		Y	Y
	Exon	Y	Y	N	N	N	N	N	N	Y	N	N	N	N	N		Y	Y
	EST Gene	N	N	N	N	Y	Y	N	N	Y	N	N	N	N	N		Y	Y
	EST Transcript	N	N	N	Y	N	Y	N	N	Y	N	N	N	N	N		Y	Y
	EST Exon	N	N	N	Y	Y	N	N	N	Y	N	N	N	N	N		Y	Y
	Prediction Transcript	N	N	N	N	N	N	N	Y	Y	N	N	N	N	N		Y	Y
	Prediction Exon	N	N	N	N	N	N	N	N	N	N	N	N	N	N		N	N
	Probeset	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	Y	Y	N		Y	Y
	Probe	N	N	N	N	N	N	N	N	Y	N	Y	N	N	N		Y	Y
	Hit	N	N	N	N	N	N	N	N	N	N	N	N	N	N		N	N
	Protein	Y	Y	N	N	N	N	N	N	Y	N	N	N	Y	N		Y	Y
	Domain	Y	Y	N	N	N	N	N	N	Y	N	N	Y	N	N		Y	Y
	Symbol	Y	Y	N	Y	Y	N	N	N	N	N	N	N	N	N		N	N

Table 1: The mappings and functions available in annmap database. Mapping queries are of the form `XXXToYYY()`, searches using genome coordinates, `XXXInRange()`, and detailed annotation `XXXDetails()`

3 Fetching Data

3.1 Retrieving all instances of a given type.

It is often useful to get a list of, for example, all genes or transcripts in the database. For this we would use functions such as `allGenes` or `allTranscripts`. All the functions of this type are defined in the documentation object `?annmap.all`.

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

```
annmapConnect( 'human66' )
```

```
Connected to annmap_homo_sapiens_66 (localhost)
Selected array 'HuEx-1_0' as a default.
```

```
> allChromosomes()
```

```
GRanges with 25 ranges and 0 metadata columns:
       seqnames         ranges strand
          <Rle>      <IRanges>  <Rle>
   [1]        1 [1, 249250621]      *
   [2]       10 [1, 135534747]      *
   [3]       11 [1, 135006516]      *
   [4]       12 [1, 133851895]      *
   [5]       13 [1, 115169878]      *
   ...      ...            ...    ...
  [21]        8 [1, 146364022]      *
  [22]        9 [1, 141213431]      *
```

```
 [23]        MT [1,      16569]      *
 [24]         X [1, 155270560]      *
 [25]         Y [1,  59373566]      *
 ---
 seqlengths:
   1 10 11 12 13 14 15 16 17 18 19  2 20 21 22  3  4  5  6  7  8  9 MT  X  Y
  NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

By default these functions all return a GRanges object (if the concept of genomic coordinates is appropriate), or a data.frame (if a genomic region doesn't exist for the object). It is also possible to just get the names of the chromosomes by passing as.vector=TRUE as a parameter:

```
> allChromosomes( as.vector=TRUE )
```

```
 [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"
```

3.1.1 Possible values for as.vector and as.data.frame

Almost all of the functions in annmap accept an as.vector parameter, and will return a vector of identifiers rather than the full table of available information. Setting as.vector='data.frame' will return a data.frame, rather than a GRanges object:

parameter		return type	
as.vector=TRUE	=>	a character vector of ids	
as.vector=FALSE	=>	A GRanges object	(default)
as.vector='data.frame'	=>	A data.frame	

All of the XXXDetails() functions and functions that map to Hits or coding regions take a parameter as.data.frame instead (where a vector would not be possible):

parameter		return type	
as.data.frame=FALSE	=>	A GRanges object	(default)
as.data.frame=TRUE	=>	A data.frame	

3.2 Getting detailed annotation based on database identifiers.

Sometimes, you will have a list of IDs for which you require more detailed annotation. You can use the ?annmapDetails methods for this.

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

```
> chr.list = c( '1', '2', '3' )
> chromosomeDetails( chr.list )
```

```
GRanges with 3 ranges and 0 metadata columns:
      seqnames          ranges strand
         <Rle>       <IRanges>  <Rle>
  [1]        1 [1, 249250621]      *
  [2]        2 [1, 243199373]      *
  [3]        3 [1, 198022430]      *
  ---
  seqlengths:
    1  2  3
   NA NA NA
```

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

```
> some.probes = probesetToProbe( '3855295' )
> probeDetails( some.probes )
                  sequence probe_hit_count
1 GTCACCCTGCAACCGTGCCACGTGG               1
2 GTCATGTGGCGCCTGTTTCGACGCC               1
3 CAGGCTCTAGGACTGCGCGATGAGC               1
4 ACCAGGTCTGGCTCGCGGCGGACGT               1
```

Note that Probe sequences inside the annmap database are not the exact physical sequences to be found on the corresponding array. Instead, we always return the sequence of the genomic target.

3.2.1 The order of things

When calling the following transformation queries (XXXToYYY, and XXXDetails), you should not rely on the order of the output being the same as the input order.

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[ elementMetadata( d )$stable_id == 'PTEN', ]`), or you can split the results based on a column (see section 3.3.3)

3.3 One thing to another

In this section, we look at the available functions for mapping between annotation items. The majority of mappings are available, and all take the form XXXToYYY[3] (Table 1)

3.3.1 Finding a gene by its symbol

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

```
> symbolToGene( 'PTEN' )
GRanges with 1 range and 9 metadata columns:
      seqnames               ranges strand |         IN1       stable_id
         <Rle>            <IRanges>  <Rle> | <character>     <character>
  [1]       10 [89622870, 89731687]      + |        PTEN ENSG00000171862
            biotype      status
        <character> <character>
  [1] protein_coding       KNOWN
                                                            description
                                                            <character>
  [1] phosphatase and tensin homolog [Source:HGNC Symbol;Acc:9588]
      db_display_name      symbol              symbol_description
          <character> <character>                     <character>
  [1]     HGNC Symbol        PTEN phosphatase and tensin homolog
          analysis_name
            <character>
  [1] ensembl_havana_gene
  ---
  seqlengths:
   10
   NA
```

[3] A full list of functions can be found by entering ?annmapTo in an R console.

Or by vector of symbols:

```
> symbolToGene( c( 'PTEN', 'SHH' ) )
```

```
GRanges with 2 ranges and 9 metadata columns:
      seqnames                 ranges strand |         IN1       stable_id
         <Rle>              <IRanges>  <Rle> | <character>     <character>
  [1]       10 [ 89622870,  89731687]      + |        PTEN ENSG00000171862
  [2]        7 [155592680, 155604967]      - |         SHH ENSG00000164690
             biotype      status
         <character> <character>
  [1] protein_coding       KNOWN
  [2] protein_coding       KNOWN
                                                         description
                                                         <character>
  [1] phosphatase and tensin homolog [Source:HGNC Symbol;Acc:9588]
  [2]                sonic hedgehog [Source:HGNC Symbol;Acc:10848]
      db_display_name      symbol             symbol_description
          <character> <character>                    <character>
  [1]     HGNC Symbol        PTEN phosphatase and tensin homolog
  [2]     HGNC Symbol         SHH                 sonic hedgehog
            analysis_name
              <character>
  [1] ensembl_havana_gene
  [2] ensembl_havana_gene
  ---
  seqlengths:
   10  7
   NA NA
```

Again, `as.vector` works as before:

```
> symbolToGene( c( 'PTEN', 'SHH' ), as.vector=TRUE )
```

```
[1] "ENSG00000171862" "ENSG00000164690"
```

3.3.2 Mapping from a gene to its transcripts

```
> gene = symbolToGene( c( 'PTEN', 'SHH' ) )
> geneToTranscript( gene )
```

```
GRanges with 10 ranges and 13 metadata columns:
       seqnames                 ranges strand |             IN1       stable_id
          <Rle>              <IRanges>  <Rle> |     <character>     <character>
   [1]       10 [ 89622870,  89731687]      + | ENSG00000171862 ENST00000371953
   [2]       10 [ 89624206,  89626806]      + | ENSG00000171862 ENST00000487939
   [3]       10 [ 89624225,  89654083]      + | ENSG00000171862 ENST00000462694
   [4]       10 [ 89685280,  89693244]      + | ENSG00000171862 ENST00000498703
   [5]       10 [ 89711956,  89721066]      + | ENSG00000171862 ENST00000472832
   [6]        7 [155592680, 155604967]      - | ENSG00000164690 ENST00000297261
   [7]        7 [155592733, 155601766]      - | ENSG00000164690 ENST00000441114
   [8]        7 [155592734, 155601766]      - | ENSG00000164690 ENST00000430104
   [9]        7 [155592744, 155601766]      - | ENSG00000164690 ENST00000435425
  [10]        7 [155599276, 155600064]      - | ENSG00000164690 ENST00000472308
                      biotype      status description      db_display_name
```

```
                 <character> <character> <character>          <character>
 [1]          protein_coding       KNOWN        <NA> HGNC transcript name
 [2]    processed_transcript       KNOWN        <NA> HGNC transcript name
 [3]    processed_transcript       KNOWN        <NA> HGNC transcript name
 [4]    processed_transcript       KNOWN        <NA> HGNC transcript name
 [5]    processed_transcript       KNOWN        <NA> HGNC transcript name
 [6]          protein_coding       KNOWN        <NA> HGNC transcript name
 [7] nonsense_mediated_decay       KNOWN        <NA> HGNC transcript name
 [8]          protein_coding    PUTATIVE        <NA> HGNC transcript name
 [9] nonsense_mediated_decay       KNOWN        <NA> HGNC transcript name
[10]    processed_transcript       KNOWN        <NA> HGNC transcript name
          symbol            symbol_description translation_start
     <character>                   <character>         <integer>
 [1]    PTEN-001 phosphatase and tensin homolog              1358
 [2]    PTEN-002 phosphatase and tensin homolog              <NA>
 [3]    PTEN-003 phosphatase and tensin homolog              <NA>
 [4]    PTEN-004 phosphatase and tensin homolog              <NA>
 [5]    PTEN-005 phosphatase and tensin homolog              <NA>
 [6]     SHH-001                 sonic hedgehog               152
 [7]     SHH-005                 sonic hedgehog                91
 [8]     SHH-003                 sonic hedgehog                91
 [9]     SHH-004                 sonic hedgehog                91
[10]     SHH-002                 sonic hedgehog              <NA>
     translation_start_exon translation_end translation_end_exon
                  <integer>       <integer>            <integer>
 [1]                7035840             186              7036264
 [2]                   <NA>            <NA>                 <NA>
 [3]                   <NA>            <NA>                 <NA>
 [4]                   <NA>            <NA>                 <NA>
 [5]                   <NA>            <NA>                 <NA>
 [6]                6965401             827              6965465
 [7]                6965509              80              6965518
 [8]                6965509             206              6965536
 [9]                6965509              71              6965551
[10]                   <NA>            <NA>                 <NA>
                 analysis_name
                   <character>
 [1] ensembl_havana_transcript
 [2]                    havana
 [3]                    havana
 [4]                    havana
 [5]                    havana
 [6] ensembl_havana_transcript
 [7]                    havana
 [8]                    havana
 [9]                    havana
[10]                    havana
---
seqlengths:
 10  7
 NA NA
```

And with `as.vector`:

```
> transcripts = geneToTranscript( gene, as.vector=TRUE )
> transcripts
 [1] "ENST00000371953" "ENST00000487939" "ENST00000462694" "ENST00000498703"
 [5] "ENST00000472832" "ENST00000297261" "ENST00000441114" "ENST00000430104"
 [9] "ENST00000435425" "ENST00000472308"
```

We have tried to keep all of the methods in annmap "type agnostic" - so this will work passing a `character` vector as the parameter in place of a `GRanges` object:

```
> gene = symbolToGene( 'PTEN', as.vector=TRUE )
> class( gene )
[1] "character"
> geneToTranscript( gene )
GRanges with 5 ranges and 13 metadata columns:
      seqnames               ranges strand |             IN1       stable_id
         <Rle>            <IRanges>  <Rle> |     <character>     <character>
  [1]       10 [89622870, 89731687]      + | ENSG00000171862 ENST00000371953
  [2]       10 [89624206, 89626806]      + | ENSG00000171862 ENST00000487939
  [3]       10 [89624225, 89654083]      + | ENSG00000171862 ENST00000462694
  [4]       10 [89685280, 89693244]      + | ENSG00000171862 ENST00000498703
  [5]       10 [89711956, 89721066]      + | ENSG00000171862 ENST00000472832
                   biotype      status description      db_display_name
               <character> <character> <character>          <character>
  [1]       protein_coding       KNOWN        <NA> HGNC transcript name
  [2] processed_transcript       KNOWN        <NA> HGNC transcript name
  [3] processed_transcript       KNOWN        <NA> HGNC transcript name
  [4] processed_transcript       KNOWN        <NA> HGNC transcript name
  [5] processed_transcript       KNOWN        <NA> HGNC transcript name
           symbol             symbol_description translation_start
      <character>                    <character>         <integer>
  [1]    PTEN-001 phosphatase and tensin homolog              1358
  [2]    PTEN-002 phosphatase and tensin homolog              <NA>
  [3]    PTEN-003 phosphatase and tensin homolog              <NA>
  [4]    PTEN-004 phosphatase and tensin homolog              <NA>
  [5]    PTEN-005 phosphatase and tensin homolog              <NA>
      translation_start_exon translation_end translation_end_exon
                   <integer>       <integer>            <integer>
  [1]                7035840             186              7036264
  [2]                   <NA>            <NA>                 <NA>
  [3]                   <NA>            <NA>                 <NA>
  [4]                   <NA>            <NA>                 <NA>
  [5]                   <NA>            <NA>                 <NA>
                 analysis_name
                   <character>
  [1] ensembl_havana_transcript
  [2]                    havana
  [3]                    havana
  [4]                    havana
  [5]                    havana
  ---
  seqlengths:
   10
   NA
```

3.3.3 The IN1 field in 'to' results

The GRanges and data.frame returned by 'to' queries, contain a column IN1. This corresponds to the initial query that led to the row being generated. If we extract the IN1 column, we get the original 2 genes that we sent to the query:

```
> elementMetadata( geneToTranscript( symbolToGene( c( 'PTEN', 'SHH' ) ) ) )$IN1
 [1] "ENSG00000171862" "ENSG00000171862" "ENSG00000171862" "ENSG00000171862"
 [5] "ENSG00000171862" "ENSG00000164690" "ENSG00000164690" "ENSG00000164690"
 [9] "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 annmap.

It can also be used with the split method to get the returned data into a list, for example, to generate a list, one element per gene, each containing the exons found in that gene:

```
> exons = geneToExon( symbolToGene( c( 'PTEN', 'SHH' ) ) )
> split( elementMetadata( exons )[['stable_id']], elementMetadata( exons )[['IN1']] )
$ENSG00000164690
 [1] "ENSE00001086614" "ENSE00001086617" "ENSE00001149618" "ENSE00001795984"
 [5] "ENSE00001760492" "ENSE00001612491" "ENSE00001676036" "ENSE00001758320"
 [9] "ENSE00001655961" "ENSE00001800592" "ENSE00001908904" "ENSE00001889423"

$ENSG00000171862
 [1] "ENSE00001456562" "ENSE00001156351" "ENSE00001156344" "ENSE00003595610"
 [5] "ENSE00001156330" "ENSE00001156327" "ENSE00003601394" "ENSE00001156315"
 [9] "ENSE00001456541" "ENSE00001813529" "ENSE00001881230" "ENSE00001926457"
[13] "ENSE00001920439" "ENSE00001837210" "ENSE00003597859" "ENSE00001893700"
[17] "ENSE00001876418" "ENSE00003657126" "ENSE00001939988"
```

3.3.4 Microarray probeset mappings

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

To select your array of choice, you can use the arrayType() method:

```
> arrayType( 'HuEx-1_0' )
Using array 'HuEx-1_0'.
```

As with annmapConnect() calling this function with no parameters will give you a menu of available arrays to choose from.

Probesets are treated like any other feature, so:

```
> exonToProbeset(geneToExon(symbolToGene('MPI')),as.vector=TRUE)
 [1] "3601956" "3601960" "3601963" "3601964" "3601965" "3601969" "3601971"
 [8] "3601959" "3601974" "3601975" "3645143" "3975448" "2746540" "2880800"
[15] "3535951" "2410077" "3655014" "3462599" "2662284" "3638759" "2766811"
[22] "3822407" "2830115" "3585206" "3825799" "3961893" "2735060" "2646089"
[29] "3471746" "3720274" "3017565" "3328152" "3248968" "2713240" "2685561"
[36] "2328589" "3415635" "3869300" "3067623" "3259813" "3707032" "3731058"
[43] "3190116" "3384517" "3235475" "2676567" "2796100" "2923872" "2591241"
[50] "3056411" "3474991" "2756434" "2813774" "2861407" "3601973" "2544232"
[57] "3336154" "2393742" "3318836" "3470332" "3841120" "3553844" "3866317"
[64] "3464331" "3665054" "3109991" "2622458" "3970147" "2396528" "3454391"
[71] "3472536" "3601957" "3601970" "3601958" "3601961" "3601966" "3601972"
[78] "3601968"
```

will find all probesets that map to exons in the gene MPI (as a vector).

Ommitting the as.vector=TRUE parameter in the call will give us the results in a data.frame. Here's the details of the first probeset from our last query:

```
> exonToProbeset(geneToExon(symbolToGene('MPI')) )[1,]
              IN1 stable_id array_name probe_count hit_score gene_score
1 ENSE00002608534   3601956   HuEx-1_0           4         1          1
  transcript_score exon_score est_gene_score est_transcript_score
1                2          2              0                    0
  est_exon_score prediction_transcript_score prediction_exon_score
1              0                           0                     0
  protein_score domain_score
1             0            0
```

As you can see, a probeset's details, comprises a set of scores. 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 hit_score of 1 means that each and every probe in the probeset hit the genome once and only once, but a score of 2 for gene_score means that one or more of the probes hits more than one gene. Since if it also has a hit score of 1, this means that the match is at a region where 2 genes are overlapping.

3.3.5 Some subtleties with probeset mappings

Probeset mappings in annmap are done in two ways:

1. Probes are mapped to the entire genome, and their match locations recorded, and
2. they are also mapped directly to cDNA sequences in order to pick up probes and probesets that fall on exon junctions.

probesetToCdnatranscript and transcriptToCdnaprobeset provide the mappings needed to retrieve these probesets.

4 Searching by genome coordinates

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:

```
> geneInRange( '7', 1000000, 1060000, 1 )
```

```
GRanges with 1 range and 8 metadata columns:
      seqnames             ranges strand |       stable_id        biotype
         <Rle>          <IRanges>  <Rle> |     <character>    <character>
  [1]        7 [1022835, 1029276]      + | ENSG00000073067 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 Symbol      CYP2W1
                                          symbol_description       analysis_name
                                                 <character>         <character>
  [1] cytochrome P450, family 2, subfamily W, polypeptide 1 ensembl_havana_gene
  ---
  seqlengths:
    7
   NA
```

will find all of the genes on the forward strand of chromosome '7' that lie between positions 1000000 and 1060000. `XXXInRange` queries also accept `data.frame`, `RangedData` and `GRanges` objects (see `?annmapRange`):

```
> # Use a data.frame as the initial parameter
> .df = data.frame( chromosome_name='7', start=1000000, end=1060000, strand=1 )
> geneInRange( .df, as.vector=TRUE )
```

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

```
> # Use a RangedData object as the initial parameter
> .rd = RangedData( space='7', ranges=IRanges( start=1000000, end=1060000 ), strand=1 )
> geneInRange( .rd, as.vector=TRUE )
```

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

```
> # Use a GRanges object as the initial parameter
> .rd = RangedData( space='7', ranges=IRanges( start=1000000, end=1060000 ), strand=1L )
> .rd = as( .rd, 'GRanges' )
> .rd
```

```
GRanges with 1 range and 0 metadata columns:
      seqnames             ranges strand
         <Rle>          <IRanges>  <Rle>
  [1]        7 [1000000, 1060000]      +
  ---
  seqlengths:
    7
   NA
```

```
> geneInRange( .rd, as.vector=TRUE )
[1] "ENSG00000073067"
```

This means that it is possible to chain InRange queries as you would most of the other queries, ie;

```
> geneToSymbol(                         # Return the gene symbols
+   exonToGene(                         # get the gene for each exon
+     exonInRange(                      # get all exons contained in the GRanges
+       symbolToGene( 'PTEN' ) ) ) ) # get a GRanges object for PTEN
ENSG00000171862 ENSG00000171862 ENSG00000171862 ENSG00000171862 ENSG00000224745
         "PTEN"          "PTEN"          "PTEN"          "PTEN"  "RP11-380G5.2"
ENSG00000224745 ENSG00000224745 ENSG00000171862 ENSG00000171862 ENSG00000171862
 "RP11-380G5.2"  "RP11-380G5.2"          "PTEN"          "PTEN"          "PTEN"
ENSG00000171862 ENSG00000171862 ENSG00000171862 ENSG00000171862 ENSG00000171862
         "PTEN"          "PTEN"          "PTEN"          "PTEN"          "PTEN"
ENSG00000213613 ENSG00000171862 ENSG00000171862 ENSG00000171862 ENSG00000171862
 "RP11-380G5.3"          "PTEN"          "PTEN"          "PTEN"          "PTEN"
ENSG00000171862 ENSG00000171862 ENSG00000171862
         "PTEN"          "PTEN"          "PTEN"
```

For example, to find all probes that target within 1Kb of the 3´ end of each gene in genes, we can first alter the range of these objects so they are just covering our region of interest:

```
> genes = geneInRange( '7', 1000000, 1060000, 1 )
> start(genes) = end( genes ) - 1000
```

Before calling probeInRange on the modified GRanges object:

```
> probeInRange( genes )
GRanges with 27 ranges and 2 metadata columns:
       seqnames               ranges strand   |                 sequence
          <Rle>            <IRanges>  <Rle>   |              <character>
   [1]        7 [1028280, 1028304]      +   | TCTGTGTTGGGGAGCGCCTGGCCAG
   [2]        7 [1028310, 1028334]      +   | AGCTCTTCCTGCTGTTTGCCGGCCT
   [3]        7 [1028334, 1028358]      +   | TCCTGCAGAGGTACCGCCTGCTGCC
   [4]        7 [1028398, 1028422]      +   | CGCCCGGGCTTTTACCATGAGGCCG
   [5]        7 [1028399, 1028423]      +   | GCCCGGGCTTTTACCATGAGGCCGA
   ...      ...                  ...    ... ...                       ...
  [23]        7 [1028980, 1029004]      +   | AGCCACTGGGGCCATGCGTATGACT
  [24]        7 [1029043, 1029067]      +   | AGGCACTGGCGCCAGAGGCTTCCTT
  [25]        7 [1029089, 1029113]      +   | AGCCCCTGAAGACAAGCAGCACTGC
  [26]        7 [1029122, 1029146]      +   | AAATGGAAACACTGACCCGGTGCGG
  [27]        7 [1029125, 1029149]      +   | TGGAAACACTGACCCGGTGCGGTGG
       probe_hit_count
             <numeric>
   [1]               1
   [2]               1
   [3]               1
   [4]               1
   [5]               1
   ...             ...
  [23]               1
  [24]               1
  [25]               1
```

```
 [26]                 1
 [27]                 1
 ---
 seqlengths:
   7
  NA
```

As with other queries, the `as.vector` parameter can be used to return more concise results:

```
> probeInRange( genes, as.vector=TRUE )
 [1] "TCTGTGTTGGGGAGCGCCTGGCCAG" "AGCTCTTCCTGCTGTTTGCCGGCCT"
 [3] "TCCTGCAGAGGTACCGCCTGCTGCC" "CGCCCGGGCTTTTACCATGAGGCCG"
 [5] "GCCCGGGCTTTTACCATGAGGCCGA" "CCGGGCTTTTACCATGAGGCCGAGG"
 [7] "GGGCTTTTACCATGAGGCCGAGGGC" "CCCACAGCTCGGACTGCTCTGGGAG"
 [9] "GTCAGCAACTGCTTCCGGTTACACC" "TGCTTCCGGTTACACCCAGGACTAC"
[11] "CTGAAGCTGCACTCCCACCCACCTA" "CTGCAGGGAGACAACGGGTGGCTGC"
[13] "GAGACAACGGGTGGCTGCATCCAGC" "GACAACGGGTGGCTGCATCCAGCCA"
[15] "CTGCATCCAGCCAGAGACAGGCGCA" "TGGGTGTCCTCAGCGTGCGAGCCCT"
[17] "GGTGTCCTCAGCGTGCGAGCCCTGC" "TGTCCTCAGCGTGCGAGCCCTGCAC"
[19] "CTCAGCGTGCGAGCCCTGCACCCCC" "ACTCCATTCCCGCTCCTGGAACACT"
[21] "TCCATTCCCGCTCCTGGAACACTTC" "TGTGCCTGGAGGCAGTCGGCCTGCA"
[23] "AGCCACTGGGGCCATGCGTATGACT" "AGGCACTGGCGCCAGAGGCTTCCTT"
[25] "AGCCCCTGAAGACAAGCAGCACTGC" "AAATGGAAACACTGACCCGGTGCGG"
[27] "TGGAAACACTGACCCGGTGCGGTGG"
```

For example, it is possible to find estGenes which overlap known genes as follows:

```
> estGeneInRange( symbolToGene( c( 'lama3', 'tp53', 'shh' ) ), as.vector=TRUE )
 [1] "ENSESTG00000004861" "ENSESTG00000004918" "ENSESTG00000032897"
 [4] "ENSESTG00000032910" "ENSESTG00000032912" "ENSESTG00000032918"
 [7] "ENSESTG00000032924" "ENSESTG00000032930" "ENSESTG00000032952"
[10] "ENSESTG00000033006" "ENSESTG00000010677"
```

4.2 Adventures with annmapRangeApply

A common task is to iterate down a `GRanges` object and perform a function per row. This functionality is provided by the method `annmapRangeApply()`.

To take a contrived example, lets write a function that takes a `GRanges` object, and returns a character vector of the form `chr: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 `annmapRangeApply` to map this down a set of transcripts, which we'll retrieve now:

```
> transcripts = geneToTranscript( symbolToGene( c( 'shh', 'tp53' ) ) )
```

Before we can do this, though, we need to tell the function which columns in the `GRanges` 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 `annmapRangeApply`: `filter` and `coerce`. The first defines the column names in the `GRanges` object we are interested in, the second, the functions we need to coerce these to the expected types[4]:

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

Once these are defined, we can then fire off our apply statement, as follows:

```
> annmapRangeApply( transcripts,
+                   contrived.function,
+                   contrived.filter,
+                   contrived.coerce )
 [1] "17:7565097"  "17:7569404"  "17:7571720"  "17:7571720"  "17:7571720"
 [6] "17:7571720"  "17:7571722"  "17:7571722"  "17:7571739"  "17:7572887"
[11] "17:7576853"  "17:7577535"  "17:7577572"  "17:7577844"  "17:7578434"
[16] "17:7578480"  "17:7578547"  "7:155592680" "7:155592733" "7:155592734"
[21] "7:155592744" "7:155599276"
```

[4]The coerce is needed because we can't always rely on R to correctly guess the data type we're expecting ('space' in a `GRanges` object will often be converted to a factor, not a character vector as required by `paste` in our `contrived.function`)

5 Affymetrix™Array Annotation

As well as providing Ensembl annotation, annmap also provides mappings for Affymetrix™arrays, as described in *An annotation infrastructure for the analysis and interpretation of Affymetrix exon array data* Genome Biology 2007, 8:R79, doi:10.1186/gb-2007-8-5-r79.

5.1 `Exonic`, `intronic`, `intergenic` and `unreliable`

Probesets in annmap 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, the obvious methods exist. For example, we can look at the probesets that are hitting around the gene for TP53:

```
> g     = symbolToGene('TP53')
> ps    = geneToProbeset( g, as.vector=TRUE )
> length(ps)
[1] 237
```

We can then see numbers of probesets that match each category:

```
> length( exonic( ps ) )
Building probeset specificity cache......done
[1] 18
> length( intronic( ps ) )
[1] 14
> length( intergenic( ps ) )
[1] 1
> length( unreliable( ps ) )
[1] 204
```

So as you can see, hitting `TP53`, we have 18 exonic, 14 intronic, 1 intergenic[5] and 204 unreliable probesets.

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

[5]This may seem odd, but note that as only a single probe in a probeset needs to be intergenic for the entire probeset to be categorised as such. So it is possible for a probeset that mostly maps to a gene to be flagged as intergenic.

```
> length( unreliable( ps, exclude=T ) )
[1] 33
```

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

```
> u.ps = sort( unreliable( ps, exclude=T ) )
> c.ps = sort( c( intronic( ps ), exonic( ps ), intergenic( ps ) ) )
> all( u.ps == c.ps )
[1] TRUE
```

5.3 UTR filtering

The function utrProbesets can be used to find those probesets that hit UTRs:

```
> gene = symbolToGene( 'TP53' )
> probesets = geneToProbeset( gene )
> utrProbesets( probesets )
  [1] "3743926" "3743927" "3743928" "3743929" "3743930" "3743931" "3743932"
  [8] "3743933" "3743938" "3743939" "3743940" "3743908" "3743926" "3743927"
 [15] "3743928" "3743929" "3743930" "3743931" "3743932" "3743938" "3743939"
 [22] "3743940" "3743908" "3743909" "3743910" "3743911" "3743912" "3743935"
 [29] "3743926" "3743927" "3743928" "3743929" "3743930" "3743931" "3743932"
 [36] "3743938" "3743939" "3743940" "3743908" "3743909" "3743910" "3743911"
 [43] "3743912" "3743935" "3743926" "3743927" "3743928" "3743929" "3743930"
 [50] "3743931" "3743932" "3743938" "3743939" "3743940" "3743908" "3743921"
 [57] "3743922" "3743923" "3743924" "3743925" "3743926" "3743927" "3743928"
 [64] "3743929" "3743930" "3743931" "3743932" "3743938" "3743939" "3743940"
 [71] "3743926" "3743927" "3743928" "3743929" "3743930" "3743931" "3743932"
 [78] "3743938" "3743939" "3743940" "3743926" "3743927" "3743928" "3743929"
 [85] "3743930" "3743931" "3743932" "3743938" "3743939" "3743940" "3743926"
 [92] "3743927" "3743928" "3743929" "3743930" "3743931" "3743932" "3743938"
 [99] "3743939" "3743940" "3743909" "3743926" "3743927" "3743928"
```

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 = geneToTranscript( gene, as.vector=TRUE )[1:6]
> transcripts
[1] "ENST00000413465" "ENST00000359597" "ENST00000504290" "ENST00000510385"
[5] "ENST00000504937" "ENST00000269305"
> utrProbesets( 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 six transcripts of TP53.

It is also possible to limit the search to only the 3' or 5' UTR:

```
> utrProbesets( probesets, transcripts, end='3' )
[1] "3743908"
> utrProbesets( 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:

```
> utrProbesets( NULL, transcripts )
```

```
  [1] "3743908" "3743909" "3743910" "3743911" "3743912" "3743913" "3743914"
  [8] "3743915" "3743916" "3743917" "3743918" "3743919" "3743920" "3743935"
 [15] "3743936" "3743908" "3743909" "3743910" "3743911" "3743912" "3743913"
 [22] "3743914" "3743915" "3743916" "3743917" "3743918" "3743919" "3743920"
 [29] "3743935" "3743936" "3743908" "3743909" "3743910" "3743911" "3743912"
 [36] "3743913" "3743914" "3743915" "3743916" "3743917" "3743918" "3743919"
 [43] "3743920" "3743935" "3743936" "3743908" "3743909" "3743910" "3743911"
 [50] "3743912" "3743913" "3743914" "3743915" "3743916" "3743917" "3743918"
 [57] "3743919" "3743920" "3743935" "3743936" "3743908" "3743909" "3743910"
 [64] "3743911" "3743912" "3743913" "3743914" "3743915" "3743916" "3743917"
 [71] "3743918" "3743919" "3743920" "3743935" "3743936" "3743908" "3743909"
 [78] "3743910" "3743911" "3743912" "3743913" "3743914" "3743915" "3743916"
 [85] "3743917" "3743918" "3743919" "3743920" "3743935" "3743936" "3743926"
 [92] "3743927" "3743928" "3743929" "3743930" "3743931" "3743932" "3743933"
 [99] "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 `codingProbesets` function:

```
> codingProbesets( NULL, transcripts )
```

```
 [1] "3743905" "3743907" "3743908" "3743909" "3743910" "3743911" "3743912"
 [8] "3743913" "3743914" "3743915" "3743916" "3743917" "3743918" "3743919"
[15] "3743920" "3743921" "3743922" "3743923" "3743924" "3743925" "3743935"
[22] "3743936" "3743907" "3743908" "3743909" "3743910" "3743911" "3743912"
[29] "3743913" "3743914" "3743915" "3743916" "3743917" "3743918" "3743919"
[36] "3743920" "3743921" "3743922" "3743923" "3743924" "3743925" "3743935"
[43] "3743936" "3743909" "3743910" "3743911" "3743912" "3743913" "3743914"
[50] "3743915" "3743916" "3743917" "3743918" "3743919" "3743920" "3743921"
[57] "3743922" "3743923" "3743924" "3743925" "3743935" "3743936"
```

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 `transcriptToUtrRange` and `transcriptToCodingRange`. Here for example is the normal details for a given transcript ENST00000417324 which falls on the reverse strand of Chromosome 1 in human:

```
> transcriptDetails( 'ENST00000417324' )
```

```
GRanges with 1 range and 12 metadata columns:
      seqnames           ranges strand |       stable_id     biotype      status
         <Rle>        <IRanges>  <Rle> |     <character> <character> <character>
  [1]        1 [34554, 36081]      - | ENST00000417324     lincRNA       KNOWN
      description      db_display_name      symbol
      <character>          <character> <character>
  [1]        <NA> HGNC transcript name FAM138A-001
                                  symbol_description translation_start
```

```
                                   <character>         <integer>
[1] family with sequence similarity 138, member A              <NA>
    translation_start_exon translation_end translation_end_exon
                 <integer>       <integer>            <integer>
[1]                   <NA>            <NA>                 <NA>
                 analysis_name
                   <character>
[1] ensembl_havana_transcript
---
seqlengths:
  1
 NA
```

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

```
> transcriptToCodingRange( 'ENST00000417324' )
```

```
GRanges with 0 ranges and 20 metadata columns:
   seqnames    ranges strand |         IN1 transcript_id   stable_id   gene_id
      <Rle> <IRanges>  <Rle> | <character>     <numeric> <character> <numeric>
   chromosome_id     biotype      status description synonym_id external_db_id
       <numeric> <character> <character> <character>  <numeric>      <numeric>
   db_display_name      symbol symbol_description translation_start
       <character> <character>        <character>         <integer>
   translation_start_exon translation_end translation_end_exon analysis_name
                <integer>       <integer>            <integer>   <character>
       phase end.phase
   <logical> <logical>
  ---
  seqlengths:
```

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

```
> transcriptToUtrRange( 'ENST00000417324' )
```

```
GRanges with 2 ranges and 4 metadata columns:
      seqnames           ranges strand |             IN1       prime     phase
         <Rle>        <IRanges>  <Rle> |     <character> <character> <logical>
  [1]        1 [34554, 36081]      - | ENST00000417324           5      <NA>
  [2]        1 [34554, 36081]      - | ENST00000417324           3      <NA>
      translated
       <logical>
  [1]      FALSE
  [2]      FALSE
  ---
  seqlengths:
    1
   NA
```

Again, these two methods take an optional 'end' parameter to specify which end of the transcript you want to work with[6]:

```
> transcriptToCodingRange( 'ENST00000417324', end='3' )
```

[6]if you pass an 'end' to transcriptToCodingRange, you are choosing which UTR to omit from the range, i.e. passing end='3' returns you the range that covers both the coding range and the 5' end, but not the 3' end.

```
GRanges with 0 ranges and 20 metadata columns:
   seqnames    ranges strand |         IN1 transcript_id   stable_id   gene_id
      <Rle> <IRanges>  <Rle> | <character>     <numeric> <character> <numeric>
   chromosome_id     biotype      status description synonym_id external_db_id
       <numeric> <character> <character> <character>  <numeric>      <numeric>
   db_display_name      symbol symbol_description translation_start
       <character> <character>        <character>         <integer>
   translation_start_exon translation_end translation_end_exon analysis_name
                <integer>       <integer>            <integer>   <character>
       phase end.phase
   <logical> <logical>
  ---
  seqlengths:
```

```
> transcriptToUtrRange( 'ENST00000417324', end='3' )
```

```
GRanges with 1 range and 4 metadata columns:
      seqnames         ranges strand |             IN1       prime     phase
         <Rle>      <IRanges>  <Rle> |     <character> <character> <logical>
  [1]        1 [34554, 36081]      - | ENST00000417324           3      <NA>
      translated
       <logical>
  [1]      FALSE
  ---
  seqlengths:
    1
   NA
```

6 annmap's local cache

You may have noticed that the first time we called exonic on page 17, annmap 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 $\approx$ 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:

```
> annmapToggleCaching()
[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:

```
> annmapToggleCaching()
[1] TRUE
```

Currently we cache the probeset specificity data used for the filters, described above, and the calls to allXXX. The cache is stored in .annmap/cache.

7 The path to the answer matters

7.1 The probeset boundary to commutativity

In general, annmap 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 = symbolToGene( c( 'PTEN', 'SHH' ), as.vector=TRUE )
> genes
[1] "ENSG00000171862" "ENSG00000164690"
> transcripts = geneToTranscript( genes, as.vector=TRUE )
> transcripts
 [1] "ENST00000371953" "ENST00000487939" "ENST00000462694" "ENST00000498703"
 [5] "ENST00000472832" "ENST00000297261" "ENST00000441114" "ENST00000430104"
 [9] "ENST00000435425" "ENST00000472308"
> genes = transcriptToGene( transcripts, as.vector=TRUE )
> genes
[1] "ENSG00000171862" "ENSG00000164690"
> geneToSymbol( genes )
ENSG00000171862 ENSG00000164690
         "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 = symbolToGene( c( 'pten', 'shh' ), as.vector=TRUE )
> genes
[1] "ENSG00000171862" "ENSG00000164690"
> probesets = exonic( geneToProbeset( genes, as.vector=TRUE ) )
> probesets
```

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

```
> genes = probesetToGene( probesets, as.vector=TRUE )
> genes
```

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

```
> geneToSymbol( genes )
```

```
ENSG00000171862 ENSG00000213613 ENSG00000164690
         "PTEN"   "RP11-380G5.3"           "SHH"
```

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

8 Utility methods

8.1 Splicing Index

For this test, we have an eSet object stored locally which contains expression data for an MCF7/MCF10A dataset around the gene 'tp53'.

```
> load( file.path( .path.package('annmap'), 'rdata', 'HuEx-1_0.tp53.expr.RData' ) )
> x.rma

ExpressionSet (storageMode: lockedEnvironment)
assayData: 240 features, 6 samples
  element names: exprs
protocolData: none
phenoData
  rowNames: ex1MCF7_r1.CEL ex1MCF7_r2.CEL ... ex2MCF10A_r3.CEL (6
    total)
  varLabels: sample group
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: huex10stv1
```

So, we can call our `spliceIndex` method, passing this data, the gene id for 'tp53' and the fact that the first 3 columns are one sample type (mcf7) and the second 3 are the other sample.

```
> spliceIndex( x.rma, symbolToGene( 'TP53' ), gps=list(1:3,4:6) )

$ENSG00000141510
                  si    p.score t.statistic    gene.av
3743908  0.112899062 0.22479731  1.43429922 0.02815739
3743909 -0.157154547 0.51591129 -0.71178196 0.02815739
3743912 -0.044769083 0.70715317 -0.40359763 0.02815739
3743913 -0.186660284 0.33324256 -1.09958985 0.02815739
3743915 -0.162931473 0.08576344 -2.26962208 0.02815739
3743917  0.126866306 0.17884429  1.62805226 0.02815739
3743918  0.056591953 0.32816552  1.11278250 0.02815739
3743919  0.062286225 0.46526044  0.80632965 0.02815739
3743920 -0.156910026 0.05743946 -2.64234332 0.02815739
3743922  0.048387256 0.57010143  0.61787261 0.02815739
3743923 -0.102274940 0.36081354 -1.03096096 0.02815739
3743924  0.008979375 0.93770685  0.08317721 0.02815739
3743925  0.006989562 0.95775111  0.05636913 0.02815739
3743928  0.074234272 0.89284542  0.14348489 0.02815739
3743936 -0.865412389 0.11450250 -2.01226730 0.02815739
```

As can be seen, this call returns a list, with one `data.frame` per gene. In this example, we only have passed a single gene for brevity.

8.2 Mapping probes into transcript coordinates

To find the location of probes relative to the mature, spliced mRNA sequence (rather than in genome coordinate space), the function `transcriptToTranslatedprobes` joins the exons in a transcript (from the `5'` end to the `3'` end), and then converts the probe locations so that they are an offset from the `5'` end of the transcript.

```
> transcriptToTranslatedprobes( c( 'ENST00000462694', 'ENST00000329958' ) )
```

```
$ENST00000462694
GRanges with 11 ranges and 3 metadata columns:
       seqnames     ranges strand   |                 sequence probe_hit_count
          <Rle>  <IRanges>  <Rle>   |              <character>       <numeric>
   [1]       10   [19, 43]      +   | AGAGATCGTTAGCAGAAACAAAAGG               2
   [2]       10   [33, 57]      +   | GAAACAAAAGGAGATATCAAGAGGA               2
   [3]       10   [46, 70]      +   | ATATCAAGAGGATGGATTCGACTTA               2
   [4]       10   [50, 74]      +   | CAAGAGGATGGATTCGACTTAGACT               2
   [5]       10   [56, 80]      +   | GATGGATTCGACTTAGACTTGACCT               2
   ...      ...        ...    ... ...                      ...             ...
   [7]       10 [137, 161]      +   | TACAGGAACAATATTGATGATGTAG               2
   [8]       10 [255, 279]      +   | AGATACTTTGTGATGTAAACTATTA               1
   [9]       10 [290, 314]      +   | CTATAATCATTTTTTGGCTTACCGT               1
  [10]       10 [305, 329]      +   | GGCTTACCGTACCTAATGGACTTCA               1
  [11]       10 [331, 355]      +   | GGGGATACAGTTCATTTGATAAGAA               1
                   IN1
           <character>
   [1] ENSE00001926457
   [2] ENSE00001926457
   [3] ENSE00001926457
   [4] ENSE00001926457
   [5] ENSE00001926457
   ...             ...
   [7] ENSE00001920439
   [8] ENSE00001920439
   [9] ENSE00001920439
  [10] ENSE00001920439
  [11] ENSE00001920439
  ---
  seqlengths:
   10
   NA

$ENST00000329958
NULL
```

Please note, that this function does not return probes that span the exon-junctions of the combined transcript sequence.

8.3 Converting between Genomic, Transcript and Protein coordinates

In annmap, there are two functions that convert between genomic coordinates and the related transcript/protein coordinates.

```
> # Given the tp53 gene
> gene        = symbolToGene( 'tp53' )
> # And the transcripts for this gene
> transcripts = geneToTranscript( gene )
> # And the proteins for this transcript
> proteins    = transcriptToProtein( transcripts )
> # get the transcript coords for the transcripts of this gene, at the start of this gene
> genomeToTranscriptCoords( start( gene ), transcripts )
```

```
GRanges with 1 range and 1 metadata column:
             seqnames       ranges strand | coord.space
```

```
                <Rle>    <IRanges>  <Rle> | <character>
  [1] ENST00000413465 [1018, 1018]      * |  transcript
  ---
  seqlengths:
   ENST00000413465
                NA
> # With as.vector=TRUE
> genomeToTranscriptCoords( start( gene ), transcripts, as.vector=TRUE )
ENST00000413465 ENST00000359597 ENST00000504290 ENST00000510385 ENST00000504937
           1018              NA              NA              NA              NA
ENST00000269305 ENST00000455263 ENST00000420246 ENST00000445888 ENST00000576024
             NA              NA              NA              NA              NA
ENST00000509690 ENST00000514944 ENST00000574684 ENST00000505014 ENST00000508793
             NA              NA              NA              NA              NA
ENST00000604348 ENST00000503591
             NA              NA
> # Get the coding range for the transcripts
> coding = transcriptToCodingRange( transcripts )
> # And then to protein coordinates
> genomeToProteinCoords( start( coding ), proteins )
GRanges with 12 ranges and 2 metadata columns:
              seqnames     ranges strand   |     frame coord.space
                 <Rle>  <IRanges>  <Rle>   | <numeric> <character>
   [1] ENSP00000269305 [394, 394]      *   |         2     protein
   [2] ENSP00000352610 [344, 344]      *   |         2     protein
   [3] ENSP00000391127 [342, 342]      *   |         2     protein
   [4] ENSP00000391478 [394, 394]      *   |         2     protein
   [5] ENSP00000398846 [347, 347]      *   |         2     protein
   ...             ...        ...    ... ...       ...         ...
   [8] ENSP00000424104 [166, 166]      *   |         0     protein
   [9] ENSP00000425104 [199, 199]      *   |         2     protein
  [10] ENSP00000426252 [128, 128]      *   |         1     protein
  [11] ENSP00000458393 [ 32,  32]      *   |         2     protein
  [12] ENSP00000473895 [143, 143]      *   |         2     protein
  ---
  seqlengths:
   ENSP00000269305 ENSP00000352610 ... ENSP00000458393 ENSP00000473895
                NA              NA ...              NA              NA
> # With as.vector=TRUE
> genomeToProteinCoords( start( coding ), proteins, as.vector=TRUE )
ENSP00000410739 ENSP00000352610 ENSP00000269305 ENSP00000398846 ENSP00000391127
            286             344             394             347             342
ENSP00000391478 ENSP00000458393 ENSP00000425104 ENSP00000423862 ENSP00000424104
            394              32             199             156             166
ENSP00000473895 ENSP00000426252
            143             128
```

There are also functions for performing translation in the opposite direction; `proteinCoordsToGenome` and `transcriptCoordsToGenome`.

```
> pos.one = transcriptCoordsToGenome( transcripts, 1, as.vector=TRUE )
> pos.one
```

```
ENST00000413465 ENST00000359597 ENST00000504290 ENST00000510385 ENST00000504937
        7579912         7579912         7578811         7578811         7578811
ENST00000269305 ENST00000455263 ENST00000420246 ENST00000445888 ENST00000576024
        7590856         7590799         7590799         7590805         7576905
ENST00000509690 ENST00000514944 ENST00000574684 ENST00000505014 ENST00000508793
        7590805         7590745         7578437         7590805         7580752
ENST00000604348 ENST00000503591
        7590805         7590745
```

And as we are on the reverse strand, all these locations should match the end of the transcripts:

```
> pos.one == end( transcripts )
```

```
ENST00000413465 ENST00000359597 ENST00000504290 ENST00000510385 ENST00000504937
           TRUE            TRUE            TRUE            TRUE            TRUE
ENST00000269305 ENST00000455263 ENST00000420246 ENST00000445888 ENST00000576024
           TRUE            TRUE            TRUE            TRUE            TRUE
ENST00000509690 ENST00000514944 ENST00000574684 ENST00000505014 ENST00000508793
           TRUE            TRUE            TRUE            TRUE            TRUE
ENST00000604348 ENST00000503591
           TRUE            TRUE
```

We can also pass `cds=TRUE` to this method to take the UTRs into account when calculating genomic location:

```
> pos.one = transcriptCoordsToGenome( transcripts, 1, as.vector=TRUE, cds=TRUE )
> pos.one
```

```
ENST00000413465 ENST00000359597 ENST00000504290 ENST00000510385 ENST00000504937
        7579912         7579912              NA              NA              NA
ENST00000269305 ENST00000455263 ENST00000420246 ENST00000445888 ENST00000576024
        7579912         7579912         7579912         7579912         7576905
ENST00000509690 ENST00000514944 ENST00000574684 ENST00000505014 ENST00000508793
        7578533         7579912              NA              NA         7579912
ENST00000604348 ENST00000503591
        7579912         7579912
```

8.4 Plotting data with `genomicPlot`

`genomicPlot` will plot the genes and exons found at a given locus.

For example, to plot the genomic region around the gene `MPI` (Figure 2):

```
> range = symbolToGene( 'MPI' )
> ranges( range ) = ranges( range ) + 5000
> genomicPlot( range )
```

We can also show the opposite strand, in one of two ways. First, passing `draw.opposite.strand =TRUE`, results in a *'washed out'* view of the opposite strand (Figure 3).

```
> genomicPlot( range, draw.opposite.strand=TRUE )
```

Alternatively setting the `strand` column from the range parameter we are passing in (Figure 4) to `'*'` results in both strands being shown.

```
> strand( range ) = '*'
> genomicPlot( range )
```

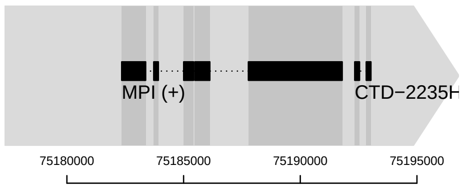

Figure 2: A genomic plot 5000bp either side of MPI.

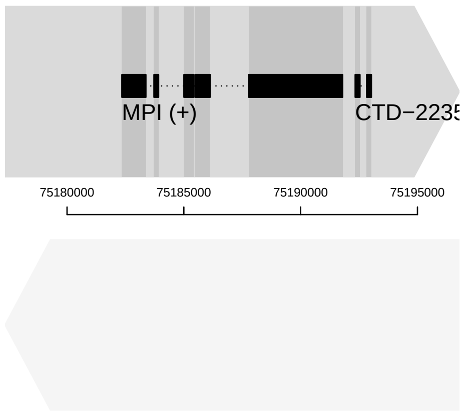

Figure 3: A genomic plot 5000bp either side of MPI with the opposite strand drawn 'washed out'.

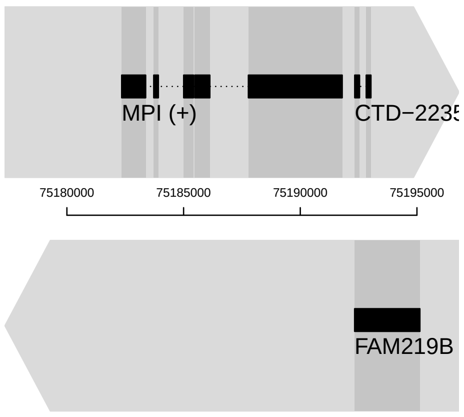

Figure 4: Both strands 5000bp either side of MPI.

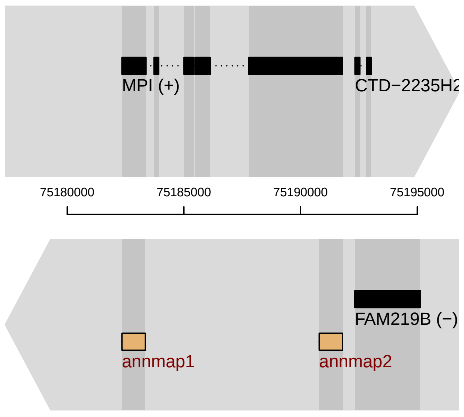

Figure 5: Both strands 5000bp either side of MPI with highlights.

It is also possible to highlight regions of the genome with your own annotation. To do this, you simply need to pass in a `data.frame` containing columns `start`, `end`, `strand` and `name`.

For example, we could add regions to the start and end locations of `MPI` but on the opposite strand and call them `annmap1` and `annmap2` (Figure 5);

```
> a1           = symbolToGene( 'MPI' )
> end( a1 )    = start( a1 ) + 1000
> a2           = symbolToGene( 'MPI' )
> start( a2 ) = end( a2 ) - 1000
> hig          = data.frame( start =c( as.integer( start( a1 ) ), as.integer( start( a2 ) ) ),
+                            end   =c( as.integer(   end( a1 ) ), as.integer(   end( a2 ) ) ),
+                            strand=c(   strandAsInteger( a1 ),      strandAsInteger( a2 )    ) *
+                            name  =c(   'annmap1',     'annmap2' ) )


> genomicPlot( range, highlights=hig )
```

8.5 Plotting NGS data with ngsBridgePlot

The ngsBridgePlot method allows NGS read data to be plotted alongside a genomicPlot. This was how Figure 1 was generated. There are many options for this plot function; here we will cover the most important ones.

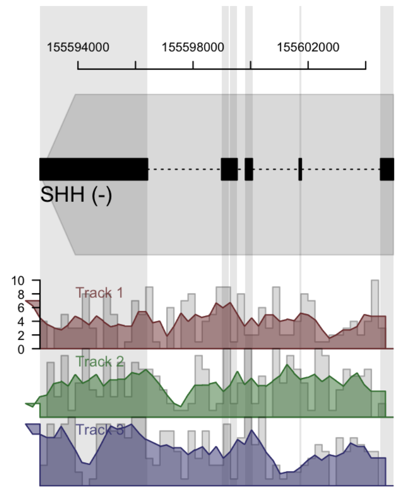

Figure 6: Dummy (in this example) NGS data shown alongside SHH.

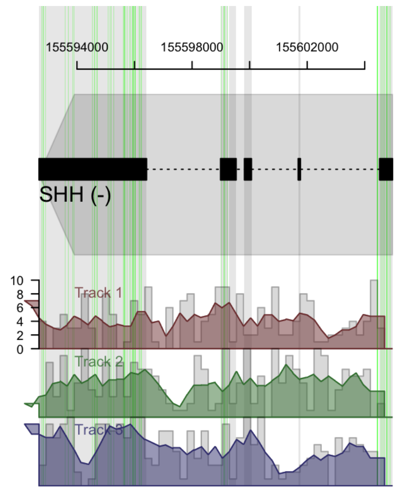

Figure 7: Dummy (in this example) NGS data shown alongside SHH with probes shown.

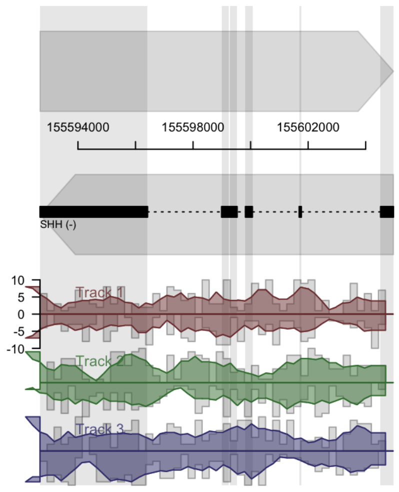

Figure 8: Dummy (in this example) NGS data shown alongside SHH (both strands) with probes shown.

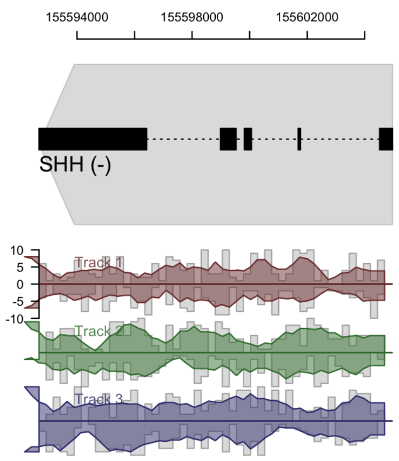

Figure 9: Dummy (in this example) NGS data shown alongside SHH. Both strands of NGS reads are shown. Exon depth plot hidden.

To plot Bamfile data alongside a region (and strand) denoted by a given gene, you can do (Figure 6):

```
> gene = symbolToGene( 'SHH' )
> ngsBridgePlot( gene, bamfileData )
```

If you want to show where the probes for a given array hit this region, you can set the probe.plot parameter to the genomicProbePlot function (Figure 7)

```
> gene = symbolToGene( 'SHH' )
> ngsBridgePlot( gene, bamfileData, probe.plot=genomicProbePlot )
```

And as we saw in the preamble to this document, we can set the strand of the given range to '*' and both strands of the region will be shown (Figure 8).

```
> gene = symbolToGene( 'SHH' )
> strand( gene ) = '*'
> ngsBridgePlot( gene, bamfileData )
```

Or, we can plot the genomicPlot for a single strand, but show the NGS data for both strands side-by-side using the `trace.match.strand` parameter and, for clarity, turn off the `exon.depth.plot` (Figure 9).

```
> gene = symbolToGene( 'SHH' )
> # trace.match.strand=FALSE would have the same effect
> ngsBridgePlot( gene, bamfileData, exon.depth.plot=NULL, trace.match.strand='*' )
```

In all these examples, the bamfileData is a synthetic object we have created for the purposes of this vignette. Its structure is:

```
> str( bamfileData )
List of 3
 $ :List of 3
  ..$ name: chr "Track 1"
  ..$ rle :List of 2
  .. ..$ +:Formal class 'Rle' [package "IRanges"] with 4 slots
  .. .. .. ..@ values         : int [1:47] 8 6 1 3 1 2 7 4 3 2 ...
  .. .. .. ..@ lengths        : int [1:47] 155592680 245 245 245 245 245 245 245 245 245 ...
  .. .. .. ..@ elementMetadata: NULL
  .. .. .. ..@ metadata       : list()
  .. ..$ -:Formal class 'Rle' [package "IRanges"] with 4 slots
  .. .. .. ..@ values         : int [1:44] 7 4 0 3 5 3 8 3 0 8 ...
  .. .. .. ..@ lengths        : int [1:44] 155592680 245 245 245 245 491 245 245 245 245 ...
  .. .. .. ..@ elementMetadata: NULL
  .. .. .. ..@ metadata       : list()
  ..$ col : chr "#804040FF"
 $ :List of 3
  ..$ name: chr "Track 2"
  ..$ rle :List of 2
  .. ..$ +:Formal class 'Rle' [package "IRanges"] with 4 slots
  .. .. .. ..@ values         : int [1:41] 9 0 6 1 7 5 1 0 1 0 ...
  .. .. .. ..@ lengths        : int [1:41] 155592680 245 245 245 491 245 245 245 245 245 ...
  .. .. .. ..@ elementMetadata: NULL
  .. .. .. ..@ metadata       : list()
  .. ..$ -:Formal class 'Rle' [package "IRanges"] with 4 slots
  .. .. .. ..@ values         : int [1:47] 2 0 8 3 9 1 5 10 4 3 ...
  .. .. .. ..@ lengths        : int [1:47] 155592680 245 245 245 245 245 245 245 245 245 ...
  .. .. .. ..@ elementMetadata: NULL
  .. .. .. ..@ metadata       : list()
  ..$ col : chr "#408040FF"
 $ :List of 3
  ..$ name: chr "Track 3"
  ..$ rle :List of 2
  .. ..$ +:Formal class 'Rle' [package "IRanges"] with 4 slots
  .. .. .. ..@ values         : int [1:46] 10 0 2 7 2 4 10 0 4 6 ...
  .. .. .. ..@ lengths        : int [1:46] 155592680 245 245 245 245 245 245 245 245 245 ...
  .. .. .. ..@ elementMetadata: NULL
  .. .. .. ..@ metadata       : list()
  .. ..$ -:Formal class 'Rle' [package "IRanges"] with 4 slots
  .. .. .. ..@ values         : int [1:44] 9 3 9 7 9 10 2 0 1 9 ...
  .. .. .. ..@ lengths        : int [1:44] 155592680 245 245 245 245 245 245 245 245 491 ...
  .. .. .. ..@ elementMetadata: NULL
  .. .. .. ..@ metadata       : list()
  ..$ col : chr "#404080FF"
```

As you can see, it is a list containing a list for each track. Each of these track lists contain the `name` of the track, `col` (its colour), and an `rle` element which contains a list of `Rle` elements (one for '+', the forward strand and one for the reverse; '-')

To generate this object from actual BAM file data there is a helper function `generateBridgeData`. Given 3 BAM files you wish to display on the plot – `data1.bam`, `data2.bam` and `data3.bam`:

```
> bamfiles = c( 'data1.bam', 'data2.bam', 'data3.bam' )
> data = generateBridgeData( symbolToGene( 'SHH' ), bamfiles )
```

Will generate the required format with sensible defaults for `name` and `col`. These defaults can be overridden by setting the `names` and `colours` parameters on the `generateBridgeData` function.