ZENBU
Data processing guide

Contents

Articles

References

Article Licenses

Data Stream Processing

Data Stream Processing

One of unique features of the ZENBU system is the ability to apply data processing and analysis on-demand at query time and as part of the visualization process. This means that raw or unprocessed data can be loaded into the ZENBU system which translates it into the internal Data Model, and then ZENBU can perform many of the data manipulations and analysis that previously required bioinformatics experts with knowledge of the unix command line and a collection of bioinformatics tools.

The data processing system is applied on a track level at query time. This means that no intermediary result needs to be stored in a database or on disk. This allows the user to modify processing parameters and immediately see the effect of the change in the visualization. It also makes the system very fast since data is processed in memory and there is no overhead of reading and writing to slow disks.

Because data processing is applied on each track, and tracks are loaded independently, there is a level of parallelism inherent in the design of the system. The processed data result generated by ZENBU on-demand can also be downloaded into data files for further analysis by external systems like R, BioConductor, or BioPython.

Data processing is controlled through a Scripting system based on chaining Processing modules together in a manner similar to digital signal processing [1]

Sorted Data Stream

The central concept of any track in the ZENBU system is that all data comes through the system as a single stream of data. This **single data stream** is often the result of **pooling** multiple data sources together.

This central data stream concept means that any object of the **Data Model** can be passed on this stream. This gives the processing and visualization systems a great deal of flexibility since all information can be made available on the data stream.

For genomic **Features**, every data stream in the system preserves a region-location sort order. When multiple sources are merged together in a Pool, the Features are "merge sorted" so that this sort order is preserved. When Features are processed by different processing modules the sort order is also preserved. By forcing all data streams to be required to follow this sort-order, it becomes very easy to write signal-processing modules which can efficiently take advantage of the fact of this sort-order. This means that many processing operation can be performed without buffering data or requiring massive amount of memory. This is one of the key features of the ZENBU system which allows it to work with Terrabytes of data yet still be able to run on modest hardware computers.

The genomic location sort order for Features appearing on the stream is as follows

- chrom_start
- chrom_end
- strand

This means that location takes priority over stand. One advantage of this sort order is that it becomes very easy to flip between stranded and strandless analysis without requiring buffering or resorting.

Scripting

ZENBU data processing scripts are an XML description language. The basic form of the script starts with an outer XML tag structure of

```
<zenbu_script>
   ...
</zenbu_script>
```
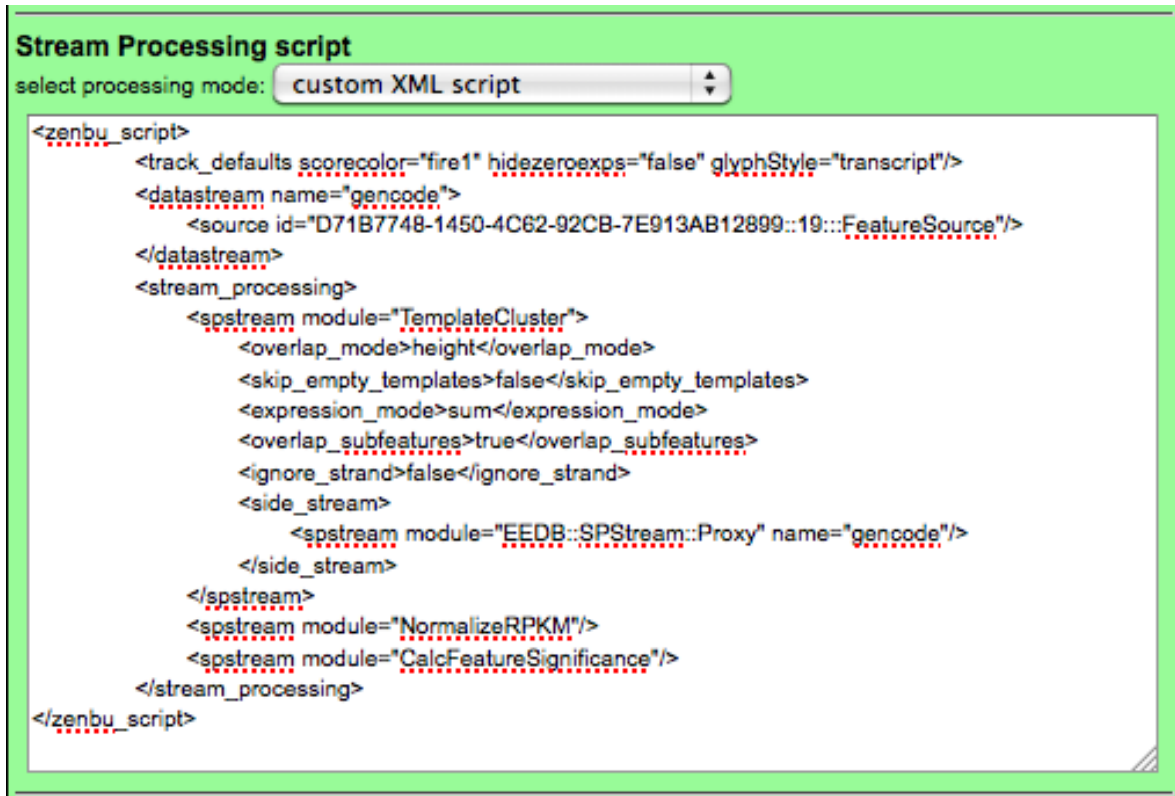

Within that structure there are several sections

- **<datastream>** : allows specification of alternate "virtual Data Source pools" for use in coordination with Proxy modules. Each different "datastream" gets its own tag section
- **<stream_processing>**: Defines the streaming-chain of processing modules which are injected between DataSource of the track and the Visualization. Data processing happens in a signal-processing style by daisy-chaining[1] multiple processing modules together. Some processing modules operate by combining data from multiple data-streams through the use of a **<side_stream>** specification inside the module configuration. In the above script example the data on the primary stream is first processed by **TemplateCluster** against a side-stream of *gencode* data sources which collates the expression into the *genocode* annotation features, followed by the second module **NormalizeRPKM** which normalized the expression, and then followed by the third module in the chain **CalcFeatureSignificance** which recalculates the combined expression of all experiments into the *significance* of each Feature on the stream.
- **<track_defaults>** : defines default options in the track configuration panel when used with "sharing a predefined script". When a predefined script with a *track_defaults* section is loaded into a track, those parameters of the "Track configuration panel" are toggled into this new default state. This makes it easy for script writers to defined a package of both processing and visualization as a "saved predefined script". Only one is <track_defaults> tag is allowed to be defined inside a script. Attribute options are:
  - *source_outmode* : sets the "feature mode" in the Data Source section
  - *datatype* : sets the "expression datatype" in the Data Source section

- *glyphStyle* : sets the visualization style
- *scorecolor*: sets the score_color by name
- *backColor*: sets the background color
- *hidezeroexps* : sets the state of the *hide zero experiments* checkbox
- *exptype* : sets the display datatype
- *height* : sets the "track pixel height" for **express** style tracks
- *expscaling* : sets the "express scale" option for **express** style tracks
- *strandless* : sets the "strandless" option for **express** style tracks
- *logscale* : sets the "log scale" option for **express** style tracks

After a script has been created and is working as desired, it can be saved and shared with other users through the **save script** button in the **track reconfigure panel**.

Please check out each of the **processing modules** below. Every module's wiki page includes an example script of how that module can be used and shows the structure of the scripting XML language. Many module pages also contain a hyper link to an active ZENBU view page as a live example of the script in action.

Processing modules

Processing is accomplished by chaining a series processing modules (or plugins) together between the pooled data source and the visualization / data download output. In addition some modules may provide for **side chaining** addition data streams into the main signal processing data stream. Side chains can be simple or complex chains of processing modules like in this case study

The processing modules can be broken down into several concept categories

Infrastructure modules

These modules provide access to additional data sources for use on side-streams

- **Proxy**: provide security-checked access to data sources loaded into ZENBU
- **FeatureEmitter:** create regular grids of features dynamically

Clustering and collation

These modules provide for high-level manipulations of data to reduce the number of features on the data stream by grouping them into related concepts.

- **TemplateCluster**: use side-chain-stream as template to collate expression.
- **UniqueFeature**: cluster and count features matching 'unique' criteria
- **Paraclu**: hierarchical clustering algorithm http://www.cbrc.jp/paraclu/

Filtering

These modules remove data from the stream based on filtering criteria

- **TemplateFilter**: use side-chain-stream as mask to filter primary stream features
- **CutoffFilter**: filter features using simple cutoff filters (high pass, low pass, band pass)
- **ExpressionDatatypeFilter**: filter expression from features based on datatype
- **FeatureLengthFilter**: filter Features based on mix/max length criteria
- **TopHits**: Filter neighborhood-regions based on *best* feature significance.
- **NeighborCutoff**: noise filtering relative to strongest signal within a neighborhood-region

Data normalization and rescaling

These modules alter the expression in a stream based on normalization or rescaling algorithms

- **NormalizeByFactor**:
- **NormalizePerMillion**:
- **NormalizeRPKM**:
- **RescalePseudoLog**:

General manipulation

These modules are general purpose *lego blocks* to manipulate objects on the stream to help with getting data in the right format for the next module in the stream.

- **CalcFeatureSignificance**:
- **CalcInterSubfeatures**:
- **StreamSubfeatures**:
- **FilterSubfeatures**: rebuild a feature/subfeature structure by filtering subfeatures
- **ResizeFeatures**:
- **MakeStrandless**:
- **RenameExperiments**:
- **FeatureRename**:

References

[1] http://en.wikipedia.org/wiki/Daisy_chain_(electrical_engineering)

Concepts

Data Sources

The ZENBU system was designed on first principle to be a collaborative OMICS data integration system where primary data is dynamically uploaded by users of the system. Because of the data processing capabilities designed into ZENBU, this uploaded data is used as *sources* for input into data processing scripts where the result of that processing can then be downloaded or visualized in the ZENBU genome browser.

The primary data which has been **uploaded** into the system are collectively referred to as **Data Sources**. When data is loaded into the system, each data file is translated into the internal ZENBU **Data Model** and grouped into **one or more annotation data sources** and/or **Expression experiment data sources** depending in the uploaded data file format and upload parameter options.

Since ZENBU is based on loading data into an abstract **data model** concept, it is important to be able to find our data after upload since there is not always a direct one-to-one mapping of *data upload file* to DataSource. To accomplish this ZENBU utilized a metadata search system. When data is uploaded, user are asked to provide a *name* and *description* of the file and its data content. By providing good descriptions, it not only allows for easily finding your data at a latter time, but also makes it easier for your collaborators to understand the content and nature of your data.

Uploadable data file formats

Before data can become available as a *Data Source* in ZENBU, it must first be uploaded into the system through one of the supported file types. The file types currently supported by ZENBU upload are:

- **BAM & SAM** sequence alignment files.
- **BED** UCSC style genome annotation files
- **GFF GFF2 GTF** Ensembl/gbrowse style genome annotation files
- **OSCTable** open format tab separated tables for genome annotation and multiple experiment expression (RIKEN/ZENBU format).

Annotation Data Sources

An **annotation data source** [also called a **FeatureSource**] is a collection of genomic features. This corresponds to a *data set* like "gene sets" or "promoter sets" or "micro array probe set" or "repeat elements". In the UCSC genome browser this is what they refer to as a **UCSC track** (i.e. data track) which was loaded from a BED file. When uploading data into ZENBU, the data in each file is mapped into a one or more **FeatureSources**.

BED is the simplest file format where each BED file maps to a single FeatureSource. There is an option when uploading a BED file to extract expression from it and optionally create an Experiment for the the file. The options include mapping the *score* onto expression or to simply count each location with an expression value of **1**.

BAM/SAM files since they always contain sequence alignments are always mapped into a single FeatureSource and single Experiment.

GFF/GTF files can be mapped into one or more FeatureSources depending on the content of the 3rd column of the file. This 3rd column encapsulates the GFF *feature* concept which is the same as the ZENBU FeatureSource concept. Every different GFF/GTF *feature* type is mapped into a different ZENBU FeatureSource.

OSCTable files are also mapped into one or more FeatureSources depending on how they are configured. The OSCTable format allows for complete flexibility to control how the data is mapped into FeatureSources upon loading into ZENBU.

Expression Experiment Data source

An expression experiment data source [also simply called a **Experiment**] is a collection of expression data. By the definition of the ZENBU data model, each expression data element is attached to a **genomic feature** which is also part of a FeatureSource. Since a genomic Feature can have many Expression data points attached, the Experiment is critical to describing the Expression.

Expression refers to a single measurement data element with an associated expression **DataType** (eg: "tagcount", "tpm", "mapquality" "score" "pvalue" "rle" to name a few). Depending on the data file used, there can be none to many Experiments associated to each uploaded data file.

BED is primary used only for genomic annotations, but ZENBU allows for optional expression experiments to be defined. If enabled at upload time, an Experiment can be created and associated with the data file. There are options to interpret the BED *score* as expression or to simply count each *bed line location* with an expression value of **1**.

GFF/GTF in the same way as BED files, ZENBU allows the score column to be mapped onto Expression and to create an optional Experiment associated with each GFF/GTF file.

BAM/SAM files since they always contain sequence alignments are always mapped into a single FeatureSource and single Experiment.

OSCTable format allows for complete flexibility to control how the data is mapped into Experiments and Expression. There can be many Experiments defined within a single OSCTable file.

Types of data which can be loaded

ZENBU was designed with a data model abstraction which allows all types of genomic annotation and expression data to be uploaded into the ZENBU system. Here are examples of the different types of experimental and analysis results data which can be loaded by users into ZENBU using the available upload file types.

Genome mapped RNA/DNA sample sequences

This class of data includes RNAseq, shortRNA, CAGE, CHiP-seq. The nature of this data is a sample of DNA or RNA which is processed by a molecular biology protocol and then sequenced. It is now very common to use next generation sequencing instruments like Illumina HiSeq2000, Illumina G3, SOLiD, 454 or Heliscope for this sequencing. This class of instrument produces millions to 100s of millions of short sequences often referred to as sequence-tags. Because of the short nature of these tags, often the best way to analyze them is to first map them onto a reference genome assembly with a program like BWA or TopHat.

The ZENBU system can directly load these genome aligned sequences with out need for additional processing. A common format for these alignments is BAM.

Genome annotations

This class of data is often the result of a bioinformatics analysis pipeline or through manual curation efforts. Common data file formats for genome annotations include BED and GFF/GTF. Since the nature of this data is descriptive, it is sometimes very useful to include descriptive metadata along with genomic location information. The OSCTable format is a highly flexible format which allows for attaching very complex metadata, expression, and numerical values onto genomic annotation features.

Microarray expression

The ZENBU system provides a means to load micro-array data. Once microarray data is loaded, it can be processed and visualized as either "expression tracks" or as "annotated-expression hybrid tracks".

Currently the loading of micro-array data is a little complicated, but we hope in the future to make this process easier for users. ZENBU currently has several micro-array probe-sets from Illumina and Affymetrix mapped onto the genome. To load micro-array expression, one needs to download one of the probe-sets into OSCTable format and then to extend the columns of the file to add the raw/normalized microarray expression for each probe. Then upload the new modified OSCTable file back into ZENBU.

Annotated Expression analysis results

The ZENBU system is able to work with very complex analysis results which can often consist of genomic locations, descriptive metadata, expression signal from multiple samples, and numerical analysis results. The OSCTable format allows any complex table of data to be mapped with column names and loaded into ZENBU. Since each analysis process/result is often unique, the flexibility of OSCTable allows the data to be loaded in its original form, rather than having to convert it into a *standard* file format and thereby have to throw away some information.

After loading complex analysis data from OSCTable files, ZENBU can process and manipulate any and all aspects of the dataset. It can be treated as simply as genomic annotation, or in complex ways with data processing and hybrid annotated-expression visualization tracks

Novel genomes

The ZENBU system was designed in the era where many novel genomes are being generated, so we made sure the system could expand into the range of 1000s of genomes. In ZENBU, genomes are treated as a "namespace" which means that the process of creating a new genome is as easy as naming it. For example if a data file names genome "human-bobsmith" chromosome "chr3" it will create it if it does not exist. The reference genome sequence and size of the chromsome are treated as additional data loaded into that "genome namespace". Once uploaded the ZENBU system will become aware of the new genome name.

Although currently not available through the web interfaces, novel genome sequences can be loaded by system administrators through ZENBU command line tools. In a near future upgrade this functionality will be available via the web interface for user upload .

Virtual DataSources - Track Data pooling

Although the DataSources are defined at data load time, the ZENBU system provides a dynamic flexible *data mixing* technology called **Data Pooling** . This allows for *virtual* DataSources to be created when configuring ZENBU tracks by mix-and-match from the uploaded physical DataSources. This provides for a great deal of flexibility both in terms of data loading and data processing without requiring external data processing and additional data loading.

Data Stream Pool

The ZENBU system allows for the dynamic creation of merged *virtual* Data Sources referred to as "data stream pools". This provides for a great deal of flexibility both in terms of data loading and data processing. With data pooling, there is no need to load new data every time a different "mix" is needed when configuring ZENBU tracks. One can simply use the data already loaded in the ZENBU system and create a new *virtual* DataSource mix.

Data pooling can be on a mix of either annotation FeatureSources or a mix of expression Experiment data sources. In both cases the *mixed* pool is a union or merging of the data.

The main advantage of data pooling is for data processing and analysis. It becomes possible to pool many experiments from many samples or across replicates for differential expression visualization and analysis. And with the data download capabilities these processed data pools can be exported into statistical systems like R and BioConductor for more advanced analysis. It also is easy to created merged annotation datasets without requiring a special upload. For example it is possible to create a merged data pool of all gene models (gencode, refseq, ensembl, ucsc known gene) in a single track.

But data pooling also can help with the data organization and the data loading process. Since ZENBU can merge data on demand, it becomes possible to organize data prior to upload at a more atomic level. For example we can keep each sequencing replicate of a sample in separate BAM files and allow ZENBU to created the *virtual mix* of all data from the same sample. This provides a high level of flexibility for being able to load the data files as they exist, rather than requiring pre-processing of the data prior to loading. This also gives the user great flexibility in creating new *groupings* of data after the data has been loaded even if the grouping was not in the original experimental design.

Examples

To better illustrate the concept of data pooling, we present several examples.

Pooling annotation FeatureSources - different mixes of repeat sets

There are many different classes of repeats in the genome. Sometimes we need to work with specific repeat class, sometimes we want to work with a broader class of repeats and sometimes we don't care about the class and are only concerned if *any* repeat is present. The data pool works very well here.

For example we have loaded the mouse mm9 repeatmasker data from UCSC where each class of repeat is mapped into a different annotation FeatureSource

```
http://fantom.gsc.riken.jp/zenbu/dex/#section=Annotation;asm=mm9;search=repeat
```

In this example we have created three different tracks with different data pooling of the repeat annotation FeatureSources.

1. repeat track is mix of all RNA-based repeats (RNA, rRNA, scRNA, snRNA, srpRNA, tRNA)
2. repeat track is mix of only LINE or SINE repeats
3. repeat track is mix of all 16 different repeat classes
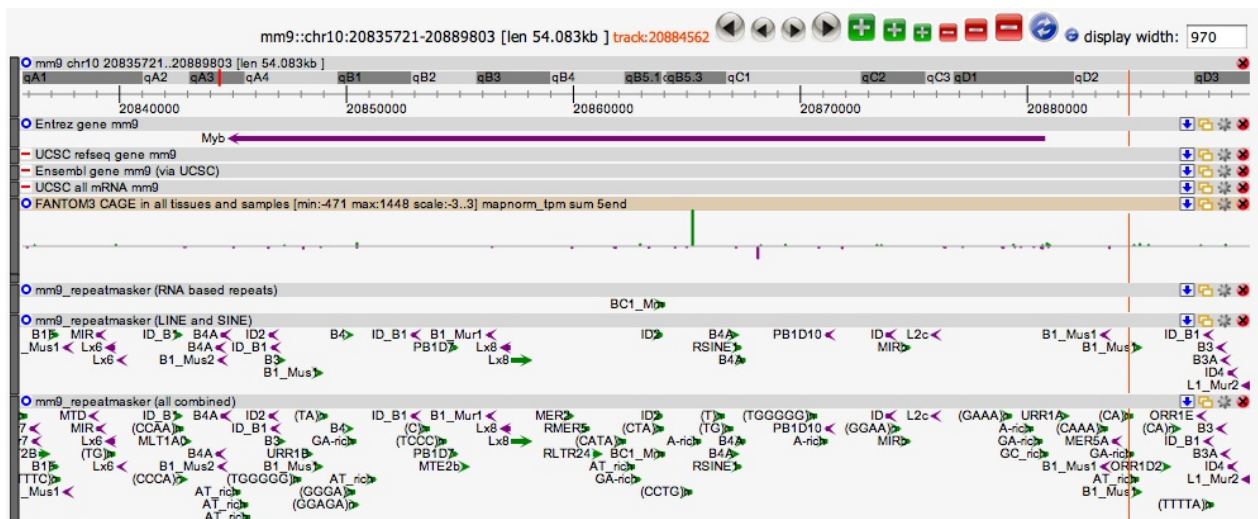
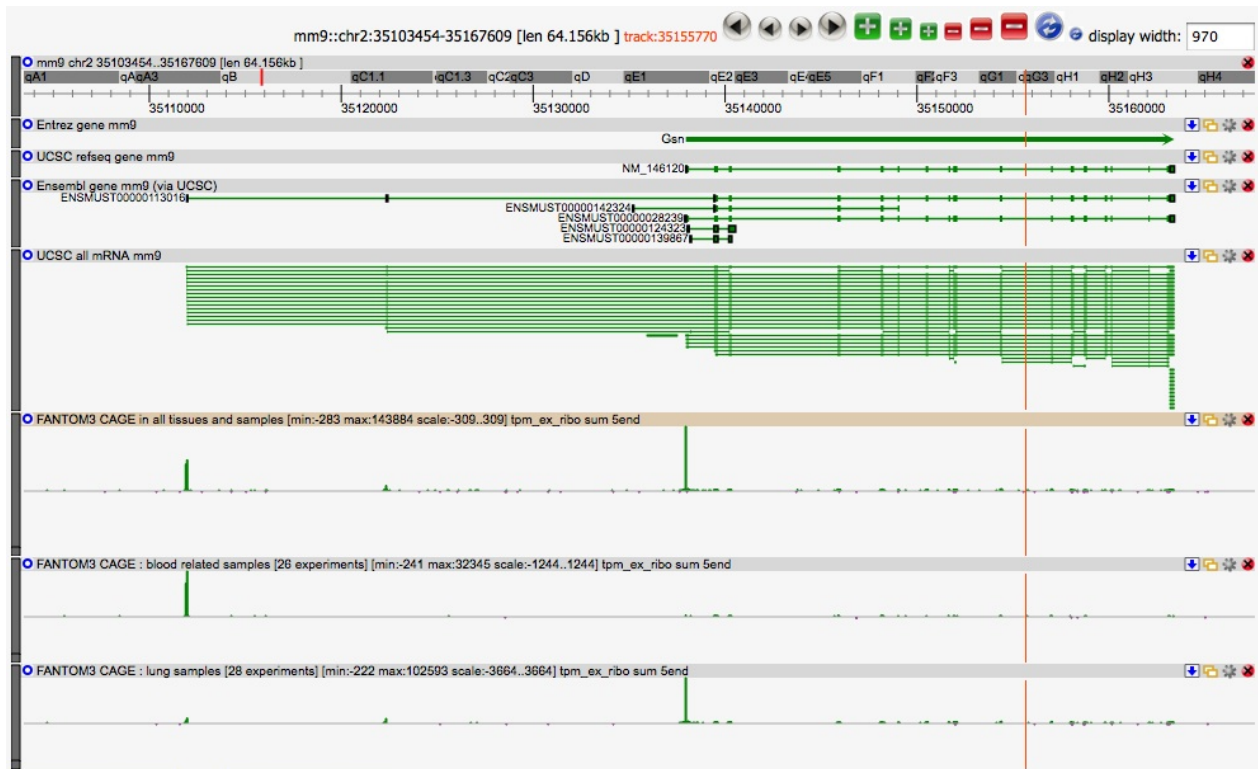link to genome browser view referenced above http:/ / fantom. gsc. riken. jp/ zenbu/ gLyphs/ ?#config=EW5pkPEsLN_uq9B-warpcD;loc=mm9::chr10:20835721..20889803

Pooling expression Experiments - FANTOM3 differential promoter expression

Differential expression is one of the key aspects when studying RNA. RNA is inherently expressed at different levels in different tissues and samples. One available RNA expression technique is called CAGE which not only records the expression level of RNAs but also identifies the RNA's 5' end location on the genome which is interpreted as the "transcription start site" for the RNA. In the FANTOM3 project, there were 465 different mouse samples which were analyzed with CAGE and sequenced. In this example we create three different tracks with different mixes of these 465 sample Experiments.

In the following view we have created three different expression tracks with different data pooling of the FANTOM3 CAGE expression Experiments.

1. expression track with virtual mix of all 465 FANTOM3 CAGE expression samples. In the view on the GSN gelsolin gene we can see two distinct CAGE expression peaks which correspond to different transcription starting sites and thus expression of different splicing isoforms of the GSN gene.
2. expression track with only 26 blood related FANTOM3 CAGE samples. In this track we can see that the blood related samples exclusively expression the left most CAGE transcription start site. By comparing to the known annotation tracks, we can see there is a long Ensembl transcript/gene (ENSMUST00000113016gene) which aligns perfectly with this CAGE peak. It is thus easy to infer that this splicing form is the one expressed in the blood.
3. expression track with only 28 lung related FANTOM3 CAGE samples. In this track we can see that the lung related samples exclusively express the right most CAGE transcription start site and thus the main splicing isoform of the GSN gene.

link to genome browser view referenced above http:/ / fantom. gsc. riken. jp/ zenbu/ gLyphs/ ?#config=OIs9ocIoGpXlzbRQEd418B;loc=mm9::chr2:35103454..35167609

Data Abstraction Model

Although the internal data abstraction model is not obviosuly exposed to the users of the system, understanding the internal model can help to understand how data is stored and processed by the system. For advanced users of the script processing system, understanding the data model is important to write your own custom processing scripts.

ZENBU internal data model

The data model is an evolution of the model first described in the FANTOM4 EdgeExpress system [1] (Genome Biol. 2009;10(4):R39. Epub 2009 Apr 19 [2]).

The ZENBU data model is composed of

- data sources (FeatureSource, Experiment, EdgeSource)
- genomic location information (Features)
- numerical expression value data (Expression)
- connections between Features (Edges)
- and descriptive metadata.
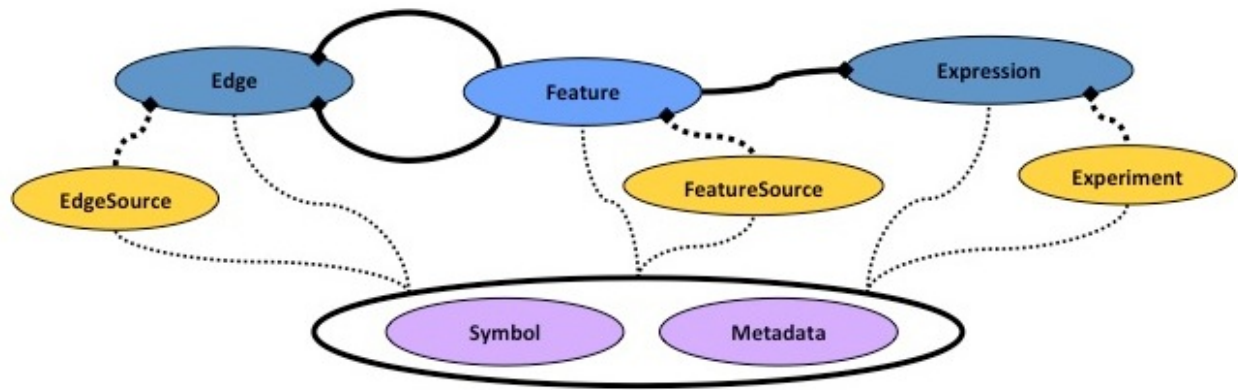
Features and associated SubFeatures

Features

The Feature is the central element in the data model.

It represents a generic object in the system. A Feature must belong to a FeatureSource. The primary attributes of a Feature include a name, a significance, and genomic coordinates.

Genomic coordinates are defined as:

- chromosome assigned to a specific species assembly
- chrom_start
- chrom_end
- strand

For Features to be visualized in the ZENBU genome browser, genomic coordinates are mandatory.

ZENBU Feature genomic coordinates are 1base-exclusive which means that chromosomes starts at 1 and features of length 1bp have the same chrom_start and chrom_end.

In addition a Feature can have Expression and Metadata attached to it.

SubFeatures

In addition a Feature can have other Features attached under it which are called **SubFeatures**.

The most common use for SubFeatures is to define exon/intron/UTR spliced gene-model details of the primary Feature, but any type of *category* can be defined for the **FeatureSources** of attached SubFeatures. For example one could define protein domains SubFeature regions of the primary Feature with different categories in addition to the exon structure.

SubFeatures are allowed to overlap each other and do not need to be exclusive.

Currently (as of version 2.5) SubFeatures cannot have another layer of SubFeatures under them.

Here is example of very complete Feature with subfeatures and expression (here displayed in a ZENBU XML export/interchange format)

```
<feature name="NM_001964" start="137801181" end="137805004"
strand="+" >
    <chrom chr="chr5" asm="hg19" ucsc_sm="hg19" ncbi_asm="GRCh37"
taxon_id="9606" length="180915260"/>
    <featuresource category="refgene" name="UCSC_hg19_refgene"
feature_count="35067"/>
    <subfeatures count="4">
```

```
        <feature category="5utr" start="137801181" end="137801451"
strand="+"/>
        <feature category="block" start="137801181" end="137801757"
strand="+"/>
        <feature category="block" start="137802446" end="137805004"
strand="+"/>
        <feature category="3utr" start="137803770" end="137805004"
strand="+"/>
    </subfeatures>
</feature>
```

Expression

Represents a single expression data element. An Expression must be attached to a Feature. In addition to the actual expression value, each Expression element has a mandatory DataType. The Expression DataType is used to describe and categorize the values so that expression from many FeatureSources and many Experiments can be pooled together for comparison. Example DataTypes include "tagcount", "tpm", "mapquality" "score" "pvalue" "rle" to name a few.

By definition each Expression data element has one Feature, one Experiment, one FeatureSource, one **DataType** and one value (number).

Edge

A connection between two Features in the system. Currently Edges are rarely used in the ZENBU system, but they have been retained from the EdgeExpress system for backward compatibility and possible future expansions.

Data Sources

These represent a collection of data of a certain class in the system and are made visible to the users in the data explorer interface.
Every DataSource has metadata describing the source which also allows for users to search and find data sets so that the data can be manipulated, and visualized.

FeatureSource

A collection of Features. Each Feature is part of only one FeatureSource.
Often used to represent a collection of annotation like "Human hg19 Entrez genes". But in addition, every file uploaded into the system is assigned a primary FeatureSource to represent that file as a collection of data.
FeatureSources can be dynamically generated by processing modules of the system to represent dynamically created Features.

Experiment

A collection of Expression data, and by connection a collection of Features.

Since a Feature can have many Expression objects attached, the Experiment is critical to describing the Expression.

EdgeSource

A collection of Edges.

This is rarely used by the current ZENBU system, but has been retained for backward compatibility to EdgeExpressDB and for future expansion capabilities.

Metadata system

Metadata is descriptive text which can be attached to any object in the ZENBU datamodel. Metadata is divided into two concepts. Metadata and Symbols.

Metadata

Metadata elements are not searchable but represent a blob of text or data.

The ZENBU system provides automatic **keyword symbol** extraction from Metadata text so that effectively to the user, the Metadata appears searchable.

In general Metadata is used for descriptive text, but it can also be XML or uuencoded data.

Symbols

Symbols are small atomic text units which can be searched.

These are often keywords or controlled vocabulary terms. Symbols can be ad-hoc or from controlled Ontologies.

Search system

The ZENBU system provides a complete metadata search system modeled on google/yahoo searching capabilities, with the addition of rigorous logic control - ***and***, ***or***, ***not*** and parenthesis *( )*

References

[1] http://fantom.gsc.riken.jp/4/edgeexpress

[2] http://dx.doi.org/10.1186/gb-2009-10-4-r39

TrackCaching System

Track Caching system

To enable fast and reliable downloading of processed data and to speed up visualization of processed track we implemented a TrackCache system based around a new binary file format called ZDX (Zenbu Data eXchangeformat).

The TrackCache is based on the concept of unique track description. Scripts are parsed such that their sole effective content (that is regardless of their formatting, the unnecessary declaration of parameters with the default value, ...) matters. This means that different people starting from scratch building tracks can generate the same "track description" (similar datasources and processing but different track title, indentation of the script, addition of comments or annotation with the script) and behind the scenes use the same trackcache

Zenbu Data eXchangeformat binary file

To enable fast and reliable downloading of processed data and to speed up visualization of processed track we implemented a TrackCache system based around a new binary file format called ZDX (Zenbu Data eXchangeformat).

The ZDX file is based on the concepts of filesystems with File-allocation-tables and inodes and file-blocks. In ZDX data is stored into znodes, but unlike filesystems where every file-block on the disk is the same size, znodes have a flexible size. The ZDX file header allows for many different subsystems of data to be stored in the same ZDX file.

The main purpose of ZDX is (like a filesysstem) to allow not only fast random access, but also to allow augmentation of data to any of the 'files' contained inside it. This is in contrast to other binary file formats in genome science.

The design also allows for the file to be always sorted even when it is partially built.

Features with expression and metadata are stored in the ZDX segments as compressed ZENBU xml using the LZ4 compression algorithm. This provides very fast compress and decompress times with still excellent compression ratios. We do not want to waste too many CPU cycles on compression/decompression. This ensures very fast read/writing of data into the ZDX segments even though it it compressed.

The main sections of how we use ZDX files in our TrackCache is

- a section dedicated to the DataSources of the track
- a pre-segmented genome (TrackCache uses a 100kb non-overlapping segment)
- a sorted Feature/Expression array attached to each segment.

Because the genome is presegmented, it is possible with ZDX to independently build each segment. As a segment is built, the data is written into a znode and appended onto the end of the ZDX file. Since everything is done with inode-like znode pointers, the actual location of the znode in the file is irrelevant. When building a TrackCache segment the TrackCacheBuilder will create a linked-list of znodes where each znode is kept around 200kb. Because the TrackCacheBuilder builds one-segment at a time and the features come out of the ZENBU streaming in sorted order, the writing into the segment is in sorted order. Therefore the ZDX file is always in sorted order, there is never a need to resort the entire file. The ZDX file built with very efficient locking so that 100s of TrackBuilders can be working on creating the same ZDX file at the same time and are only limited by the disk performance of the system.

ZDX binary files inherently enable Map/Reduce parallelization

Because of the virtual file-system like approach in ZDX, we are able to randomly build different parts of the file at the same time.

Since TrackCache ZDX has a presegmented genome, it naturally enables MapReduce style building of the complete genome. And the order of segment building does not matter. The TrackCacheBuilders use the ZENBU API for data streaming and data processing so generate the same result as the webservices.

eHive based system of autonomous-agent and work-claim design

The TrackCacheBuilders follow an autonomous-agent and work-claim design originally developed in the eHive system. TrackCacheBuilders do not need to be told what to do, they check a black-board database (like ehive) for trackcache's which are unbuilt and for user requests for region building. Once they have initialized to a particular TrackCache, they can either build user-requested segments or randomly pick an unbuilt segment. Like eHive the autonomous-agent TrackCacheBuilder workers first lock-and-claim a segment (fast no-race condition) and then proceed to build the segment at whatever pace the dataprocessing allows. This enables 100s-1000s of workers to simultaneously work on the same ZDX TrackCache without colliding (each segment is built only once). This is very efficient and completely autonomous. Because the granularity of genome TrackCache building is on the 100kilobase segment size, the system has very good latency between when a user makes a request for a segment to be built and when a free TrackCacheBuilder worker can finish it's current segment and pick up the job request to build another segment.

Just like eHive workers are given a limited lifespan before they "die" and are reborn. This also gives a layer of fault tollerence to the system (like eHive). If worker dies mid-build the segment is labeled as still mid-build with the workers process-ID, it is possible to identify the failure and reset the segment so that another worker can build it. The workers have failure code built-in so many fail-states are caught and the worker can reset the segment before needing to die. In addition all sorts of building stats are recorded for each segment. number of features, build time, worker processID, host machine name... Because of the design of the system it is very easy to have a cluster of computers running 100s of TrackCacheBuilder workers to enable high-degree of parallelization for TrackCacheBuilding. DataDownload is enabled once the requested segments have been built. This means that download can be enabled without requiring the track for the whole genome to have been build. This enhances the user-response experience.

This provides the complete flexibility of the ZENBU data processing and data pooling system.

Because the result of processing is stored in the ZDX TrackCache as ZENBU datamodel XML, it is read back out of the cache fully intact and this able to be reused for further ZENBU data processing and output. This enables the same track cache to download data into many different export formats (bed, gff, osctable). This allows for the web interface to provide much flexibility on data download and still use the same TrackCache/ZDX. And the same TrackCache/ZDX can be used for fast data query for the track visualization system and really enhances the user experience of using the genome browser.

Processing modules

Proxy

Data Stream Processing > Processing Modules > Infrastructure Modules

Description

The **Proxy** a special place holder processing module designed to work in coordination with the <datastream> section of the ZENBU scripting system.

Each <datastream> has a *name* attribute and a pool of data sources with tag <source>. Each data source is defined by their ZENBU system *id* . The other attributes of each <source> are ignored, but can be helpful for script writers as *comments*. Here is an example of a data stream pool of 4 RNAseq experiments from the Encode project from HepG2 cells.

```
<datastream name="encode_wold_hepg2">
   <source id="904A696A-62EC-4665-85B9-4F92DDFA9814::2:::Experiment" platform="RNA-seq"/>
   <source id="8EB257B8-6B26-4DB7-8470-07A708EC7CEF::2:::Experiment" platform="RNA-seq"/>
   <source id="A80763D0-F12C-449D-AFEA-288BEBE55C4A::2:::Experiment" platform="RNA-seq"/>
   <source id="74E98401-90F9-4FE4-B534-2AC4D3955753::2:::Experiment" platform="RNA-seq"/>
</datastream>
```

The matching Proxy for this datastream in a script would look like this

```
...
<spstream module="Proxy" name="encode_wold_hepg2"/>
...
```

By separating the data sources and proxy place-holders it is possible to provide

- security checking that the current user is allowed access to the data sources defined in the <datastream> sections
- allows the pooled data sources to be reused in different sections of the same script by placing multiple Proxy modules with the same *name*.
- makes it easy to copy/paste commonly used <datastream> blocks between different scripts.

Datastream attributes

Proxy and <datastream> are a special module pairing in the ZENBU script system and use these attributes to control the nature of the data on the datastream.

- **name** : name of the <datastream> which will be injected in place of this proxy at query time. Name must match between a <datastream> definition for a Proxy to correctly initialize.
- **output** : defines the level of data which will be provided on this datastream. By limiting the level of data loading, performance can be increased. Valid values are:
  - *full_feature* : Features are loaded with all available data -- genome coordinates, name, subfeatures, expression and feature metadata.
  - *simple_feature* : Features are loaded with only genome coordinates and names. No subfeatures, nor expression nor feature metadata. Default if no specified.

- *subfeature* : Features are loaded with -- genome coordinates, name, subfeatures. No expression nor metadata.
- *expression* : Features are loaded with -- genome coordinates, name, expression. No subfeatures nor metadata.
- *skip_metadata* : Features are loaded with -- genome coordinates, name, subfeatures, and expression. No metadata.
- *skip_expression* : Features are loaded with -- genome coordinates, name, subfeatures, and metadata. No expression.
- **datatype** : defineds the expression datatype for the datastream. If not specified, no expression will be available on the datastream.

Getting datastream xml definitions from tracks

The easiest way for a user to get the XML for a Proxy Datastream pool to use an already existing track which has been configured with the Data Source they are interested in.

For example below we can see a Gencode annotation track which we will use as a Proxy datastream to collate expression.
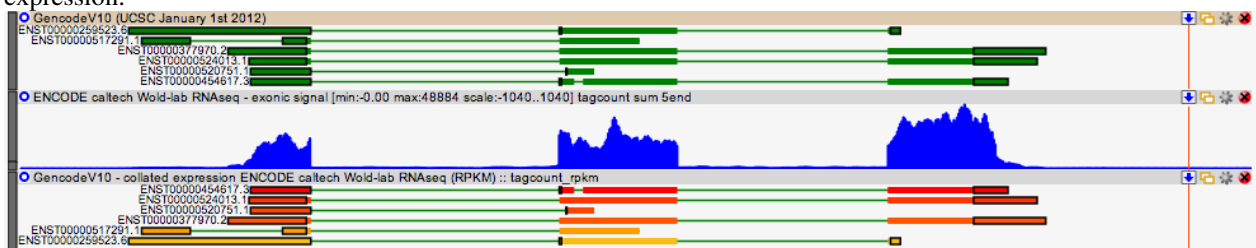
Access the Track Reconfiguration panel and then select the **datastream xml** control which will bring up a pop-up panel with the XML datastream definition for this track which can be copy-pasted into your scripts. Please note that the *name* of the datastream is an arbitrary track-number and it is best to rename the datastream-pool to something easier to remember.

Example

This is a script which incorporates a **Proxy** / TemplateCluster to collate expression into Gencode V10 gene models. The expression is then normalized via the NormalizeRPKM normalization module. The script finishes with CalcFeatureSignificance so that the Features can be displayed via score-coloring.

```
<zenbu_script>
      <datastream name="gencode" output="full_feature">
            <source id="D71B7748-1450-4C62-92CB-7E913AB12899::19:::FeatureSource"/>
      </datastream>
      <stream_queue>
            <spstream module="TemplateCluster">
                  <overlap_mode>height</overlap_mode>
                  <skip_empty_templates>false</skip_empty_templates>
                  <expression_mode>sum</expression_mode>
                  <overlap_subfeatures>true</overlap_subfeatures>
                  <ignore_strand>true</ignore_strand>
                  <side_stream>
                        <spstream module="Proxy" name="gencode"/>
                  </side_stream>
            </spstream>

            <spstream module="NormalizeRPKM"/>

            <spstream module="CalcFeatureSignificance"/>
      </stream_queue>
</zenbu_script>
```

Here is a ZENBU view showing this script in use

http://fantom.gsc.riken.jp/zenbu/gLyphs/#config=vtPXLwwqO9KjD1YYWqCMGD

FeatureEmitter

Data Stream Processing > Processing Modules > Infrastructure Modules

Description

The **FeatureEmitter** processing module generates a denovo grid of features on the genome. It is primarily used along with TemplateCluster to create regular gridded histogram-like expression on a genome. It can output features on the region of query or the whole genome. The output are simple features with coordinates, and a name. The granularity of the grid is controlled using either the *num_per_region* or the *width* parameter

Parameters

- **<dynamic/>** : specify that the width of generated grid of features is dynamically calculated based on the genomic region queried and the display_width of the visualization. This is the default behaviour.
- **<width>** : specify the desired **width** of the grid of features. for a specified region query, it will generate this many total features and adjust the feature sizes accordingly. If <width> is specified, the *dynamic* behaviour is disabled and features will always be generated at this width.
- **<fixed_grid>** : for a region query, realign the start so that it falls on a regular grid which are integer multiples of the feature width. values are true/false
- **<both_strands>** : defines if the grid of features should be generated on both strands (two for each location) or if it will generate strandless features. values are true/false

Example

The expression binning GUI provides for a convenient graphical interface to display data as a wiggle plot / fixed-grid-binned histogram.

Often after a chain of processings, it is desirable to display the data in a similar fashion.

Below is an example of the section to be added as the final step of the processing chain to attain the same results as the expression binning GUI with parameters *"overlap mode:area"; "expression binning:sum"; "process ignoring strand:unchecked"; "overlap via subfeatures:checked"* .

```
<zenbu_script>
   <stream_processing>

              ... your spstream modules chain here ...

           <spstream module="TemplateCluster">
                 <overlap_subfeatures>true</overlap_subfeatures>
                 <side_stream>
                       <spstream module="FeatureEmitter">
                              <dynamic/>
                              <both_strands>true</both_strands>
                        </spstream>
                 </side_stream>
           </spstream>

      </stream_processing>
</zenbu_script>
```

This script combines a FeatureEmitter with TemplateCluster to create a regular grid of features at a 100base resolution and collates expression evenly into each overlapping "bin".

```
<zenbu_script>
   <stream_processing>
           <spstream module="TemplateCluster">
                   <overlap_mode>height</overlap_mode>
                   <expression_mode>sum</expression_mode>
                   <overlap_subfeatures>true</overlap_subfeatures>
                   <ignore_strand>true</ignore_strand>
                   <side_stream>
                           <spstream module="FeatureEmitter">
                                   <width>100</width>
                                   <fixed_grid>true</fixed_grid>
                                   <both_strands>false</both_strands>
                           </spstream>
                   </side_stream>
           </spstream>
      </stream_processing>
</zenbu_script>
```

Example ZENBU view showing this script in use

http:// fantom. gsc. riken. jp/ zenbu/ gLyphs/ #config=bUULzhgRIBkifTt2sXZEEB;loc=hg19::chr8:128746973. . 128755020

TemplateCluster

Data Stream Processing > Processing Modules > Clustering and collation Modules

Description

The **TemplateCluster** processing module takes a stream of template features on a side stream and performs overlap comparison against features with expression on the primary data stream. When an overlap occurs, expression is collated from the primary-stream feature into the overlapping secondary stream feature. The output of the this module are template features with expression values.

Parameters

- **<side_stream>** : data source definition for features to be used as the templates for collation. This can be as simple as a **FeatureEmitter** or **Proxy** or it can be a processed stream of features.

- **<overlap_mode>** : defined how overlap calculation is performed between features on the primary stream and features on the side stream. possible values are :

  - **area** : if primary stream features overlap multiple templates, expression is evenly divided among the templates so that total counts remain the same as the input stream. Visually this creates an affect where by the expression correlates to the "area on the curve" of the feature or the number of pixels.
  - **height** : if primary stream features overlap multiple templates, expression is equally copied/collated into all template featues. Visually this gives the effect whereby the height of the resultant feature represents the collated expression, but the total sum of expression across output features is no longer preserved.
  - **5end** : the primary stream feature is compressed to the 5' end and overlap is compared against that single base location.
  - **3end** : the primary stream feature is compressed to the 3' end and overlap is compared against that single base location.

- **<expression_mode>** : defines how expression within matching Experiments are collated together. Possible values are:

  - **sum** : sum the expression between multiple primary stream features for each matching experiment into the template feature
  - **min** : calculate the minimum expression value between multiple primary stream features for each matching experiment
  - **max** : calculate the maximum expression value between multiple primary stream features for each matching experiment
  - **count** : count the number of primary stream features for each matching experiment overlapping the template feature.
  - **mean** : calculate the average expression value among primary stream features for each matching experiment overlapping the template feature

- **<ignore_strand>** : ignore strand specificity when comparing features between the primary and template streams. Enable by setting to **true**.

- **<overlap_subfeatures>** : if features contain subfeatures (eg like transcript gene models) setting this option to **true** while require that the subfeatures overlap each other in order to trigger collation of expression. If one of the features does not have subfeatures then the genomic bounds of the feature are used in the overlap calulation. If both features have subfeatures then it must be a subfeature to subfeature overlap to trigger collation.

- **<skip_empty_templates>** : if set to *false* templates with zero expression are retained. default behaviour is that templates which do not collate expression are removed from the stream.

Example1 : combining a Proxy module (co-localization with selected regions)

In order to quantify the amount of signal co-localized with particular regions, TemplateCluster can be used in combination with a Proxy module defining the regions of choice (in the case below all the regions corresponding to Entrez genes -- on either the human, mouse or rat genomes --).

```
<zenbu_script>
  <datastream name="entrez" output="simple_feature">
    <source id="0583D02E-BA10-11DE-B45C-8D369A8382FD::50:::FeatureSource" name="Entrez_gene_mm9"/>
    <source id="0583D02E-BA10-11DE-B45C-8D369A8382FD::31:::FeatureSource" name="Entrez_gene_hg18"/>
    <source id="B1880D44-F935-11DF-82E8-6158894DF986::15:::FeatureSource" name="Entrez_gene_hg19"/>
    <source id="0583D02E-BA10-11DE-B45C-8D369A8382FD::47:::FeatureSource" name="Entrez_gene_rn4"/>
  </datastream>
  <stream_stack>
   <spstream module="TemplateCluster">
    <ignore_strand value="false"/>
      <side_stream><spstream module="Proxy" name="entrez"/></side_stream>
    </spstream>
  </stream_stack>
</zenbu_script>
```

Example2 : combining with a FeatureEmitter module (strand-aware grid binning)

One of the most common use of TemplateCluster is in combination with FeatureEmitter. This script combines a FeatureEmitter with TemplateCluster to create a regular grid of features at a "screen resolution" of 970 separate "bins". This will always generate the same number of output feature/bins irrespective of the region query size. This is useful for display purposes. Input expression is collates evenly into each overlapping "bin".

```
<zenbu_script>
   <parameters>
      <source_outmode>skip_metadata</source_outmode>
      <skip_default_expression_binning>true</skip_default_expression_binning>
   </parameters>
   <stream_stack>
      <spstream module="TemplateCluster">
            <overlap_mode>height</overlap_mode>
            <expression_mode>sum</expression_mode>
            <ignore_strand>false</ignore_strand>
            <overlap_subfeatures>true</overlap_subfeatures>
            <side_stream>
                  <spstream module="FeatureEmitter">
                        <num_per_region>970</num_per_region>
                        <fixed_grid>true</fixed_grid>
                        <both_strands>true</both_strands>
                  </spstream>
```

```
                        </side_stream>
    </spstream>
</zenbu_script>
```

Example3 combining with a FeatureEmitter module (strand-less grid binning)

This script combines a FeatureEmitter with TemplateCluster to create a regular grid of **strandless** features at a 100base resolution and collates expression evenly into each overlapping "bin".

```
<zenbu_script>
    <stream_queue>
            <spstream module="TemplateCluster">
                    <overlap_mode>height</overlap_mode>
                    <expression_mode>sum</expression_mode>
                    <overlap_subfeatures>true</overlap_subfeatures>
                    <ignore_strand>true</ignore_strand>
                    <side_stream>
                        <spstream module="FeatureEmitter">
                                <width>100</width>
                                <fixed_grid>true</fixed_grid>
                                <both_strands>false</both_strands>
                        </spstream>
                    </side_stream>
            </spstream>
        </stream_queue>
</zenbu_script>
```
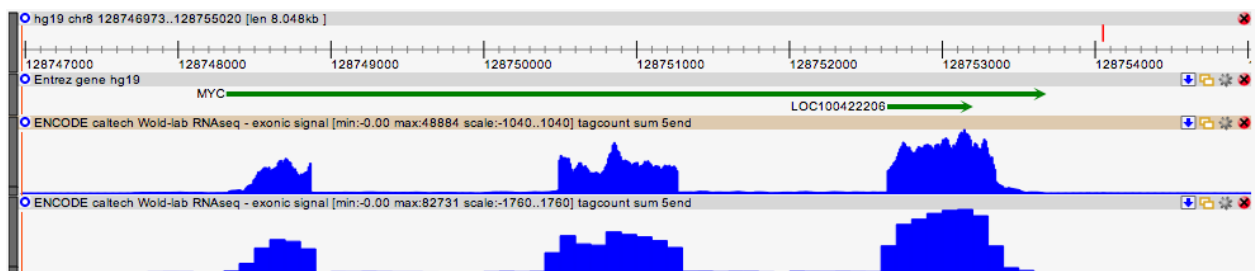
Example ZENBU view showing this script in use

http://fantom.gsc.riken.jp/zenbu/gLyphs/#config=bUULzhgRIBkifTt2sXZEEB;loc=hg19::chr8:128746973..128755020

UniqueFeature

Data Stream Processing > Processing Modules > Clustering and collation Modules

Description

The **UniqueFeature** processing module is clustering style module which combines Features which match a **unique** or **identical** criteria.

The basic criteria for Features being unique is that they share the same genomic location (chrom/start/end) including the same strand.

Parameters

- **<ignore_strand>** : if *true* then features can be on different strands, but still be consider **identical** as long as they have the same start/end/chrom location.
- **<match_category>** : if *true* then features must also have same FeatureSource category to be classified as **identical**, but do not need to be from exact same FeatureSource.
- **<match_source>** : if *true* then features must also have same FeatureSource to be classified as **identical**.
- **<expression_mode>** : defines how expression within matching Experiments between unique features are collated together. Possible values are:

  - **sum** : sum the expression between multiple primary stream features for each matching experiment into the unique feature
  - **min** : calculate the minimum expression value between multiple primary stream features for each matching experiment
  - **max** : calculate the minimum expression value between multiple primary stream features for each matching experiment
  - **count** : count the number of primary stream features for each matching experiment overlapping the unique feature.
  - **mean** : calculate the average expression value among primary stream features for each matching experiment overlapping the unique feature

Example

This is a script which incorporates a CalcInterSubfeatures, StreamSubfeatures, UniqueFeature followed by CutoffFilter to generate a set of **unique introns** which can be displayed with *score coloring*.

```
<zenbu_script>
     <stream_queue>
          <spstream module="CalcInterSubfeatures"/>
          <spstream module="StreamSubfeatures">
               <category_filter>intron</category_filter>
          </spstream>
          <spstream module="UniqueFeature">
               <ignore_strand>true</ignore_strand>
          </spstream>
          <spstream module="CutoffFilter">
               <min_cutoff>2</min_cutoff>
          </spstream>
```

```
        </stream_queue>
</zenbu_script>
```

second variation on this script using the internal unique-feature controls of the StreamSubfeatures rather than an external UniqueFeature module

```
<zenbu_script>
      <stream_queue>
            <spstream module="CalcInterSubfeatures"/>
            <spstream module="StreamSubfeatures">
                  <category_filter>intron</category_filter>
                  <unique>
                        <ignore_strand>true</ignore_strand>
                  </unique>
            </spstream>
            <spstream module="CutoffFilter">
                  <min_cutoff>2</min_cutoff>
            </spstream>
      </stream_queue>
</zenbu_script>
```

Example ZENBU view showing this script in use

http://fantom.gsc.riken.jp/zenbu/gLyphs/#config=X_wwpjePN9Qi8bqoMur8TB;loc=hg19::chr8:128746973..
128755020

Paraclu

Data Stream Processing > Processing Modules > Clustering and collation Modules

Description

The **Paraclu** processing module is a porting into the ZENBU stream processing system of the Paraclu program of Martin Frith http://www.cbrc.jp/paraclu/.Paraclu finds clusters in data attached to sequences. It was first applied to transcription start counts in genome sequences (see citation below), but it could be applied to other things too.

Paraclu is intended to explore the data, imposing minimal prior assumptions, and letting the data speak for itself. One consequence of this is that paraclu can find clusters within clusters. Real data sometimes exhibits clustering at multiple scales: there may be large, rarefied clusters; and within each large cluster there may be several small, dense clusters.

The ZENBU implementation reproduces the hierarchical clustering of the original paraclu, plus the paraclu-cut.sh filter/selection process. In addition ZENBU implements another filter/section method based on the *best density under max_cluster_length.*

Because the Paraclu algorthim was designed for a genome 1bp resolution signal strength input, it is important to follow the script example below where a combination FeatureEmitter/TemplateCluster is prepended before Paraclu.

Parameters

- **<min_cutoff>** : the minimum value of **Features** which are allowed to contribute to the paraclu analysis.
- **<stability>** : the min stability parameter of paraclu-cut.sh.

  Each paraclu cluster has a minimum and maximum density (for an explanation of "density", please consult the paraclu publication mentioned in the citation section). Briefly, the greater the fold-change between minimum and maximum density, the more prominent the cluster, and the less likely that it is due to chance fluctuations in the data. Stability is defined as (maximum density / minimum density).

  Clusters with less a *stability* smaller than the value defined by this parameter are not subdivided.
- **<max_cluster_length>** : parameter of paraclu-cut.sh.

  Clusters greater than max_cluster_length are always sub-divided

Example

This is a complex script which incorporates a FeatureEmitter / TemplateCluster expression histogram binning with de-novo clustering via Paraclu followed by several filtering steps including NeighborCutoff, CutoffFilter, and a final FeatureLengthFilter to remove very tiny clusters.

```
<zenbu_script>
    <stream_queue>
        <spstream module="TemplateCluster">
            <overlap_mode>area</overlap_mode>
            <expression_mode>sum</expression_mode>
            <side_stream>
                <spstream module="FeatureEmitter">
                    <width>1</width>
                    <fixed_grid>true</fixed_grid>
                    <both_strands>true</both_strands>
                </spstream>
            </side_stream>
        </spstream>

        <spstream module="Paraclu">
            <min_cutoff>10</min_cutoff>
            <stability>0</stability>
            <max_cluster_length>100</max_cluster_length>
        </spstream>

        <spstream module="CalcFeatureSignificance"/>

        <spstream module="NeighborCutoff">
            <ratio>300</ratio>
            <distance>100</distance>
        </spstream>

        <spstream module="CutoffFilter">
            <min_cutoff>100</min_cutoff>
        </spstream>
```

```
            <spstream module="FeatureLengthFilter">
                  <max_length>50</max_length>
            </spstream>


      </stream_queue>
</zenbu_script>
```

Example ZENBU view showing this script in use with shortRNA RNAseq alignment data showing potentially novel microRNA clusters. This example shows ParaClu clustering followed by different levels of post-filtering in different tracks.

http:// fantom. gsc. riken. jp/ zenbu/ gLyphs/ #config=jqr7D6J2PxMrOdqTg8glvD;loc=hg19::chr19:49990236. . 49997699

Citation

A code for transcription initiation in mammalian genomes [1], *MC Frith, E Valen, A Krogh, Y Hayashizaki, P Carninci, A Sandelin, Genome Research 2008 18(1):1-12*

References

[1] http://genome.cshlp.org/content/18/1/1.long

TemplateFilter

Data Stream Processing > Processing Modules > Filtering Modules

Description

The **TemplateFilter** processing module takes a stream of template features on a side stream and performs overlap comparison against features on the primary data stream. When an overlap occurs, the primary stream primary-stream feature is either passed through this filter (default behaviour) or blocked based on this module's parameter settings.

Parameters

- **<side_stream>** : data source definition for features to be used as the templates for overlap comparison. This is most often a **Proxy** for another set of defined annotations, but it can also be a processed stream of features.

- **<overlap_mode>** : defines how overlap calculation is performed between features on the primary stream and features on the side stream. possible values are :
  - **area** : the full lenght of both features are used in the comparison.
  - **5end** : the primary stream feature is compressed to the 5' end and overlap is compared against that single base location.
  - **3end** : the primary stream feature is compressed to the 3' end and overlap is compared against that single base location.

- **<ignore_strand>** : ignore strand specificity when comparing features between the primary and template streams. Enable by setting to **true**.

- **<inverse>** : inverses the filtering process. if set to *true* then overlaps are blocked not passed through. if set to *false* then overlaps are allowed to pass through.

- **<overlap_subfeatures>** : if features contain subfeatures (eg like transcript gene models) setting this option to **true** will require that the subfeatures overlap each other. If one of the features does not have subfeatures then the genomic bounds of the feature are used in the overlap calulation. If both features have subfeatures then it must be a subfeature to subfeature overlap.

- **<label_overlaps>true</label_overlaps>** : if activated, the primary_name of the overlapping side-stream features are appended into the metadata of the primary stream features. This can be useful when downloading tracks into GFF or OSCtable formats when feature-metadata is enabled.

Examples

In this example, we use **TemplateFilter** with Gencode transcripts as a side stream to filter the datastream (in the linked View below, RNA-seq from ENCODE) for signal overlapping exons (by enforcing 'overlap_subfeatures') . The histogram is then recreated thru combination of a FeatureEmitter with TemplateCluster to create a regular grid and collates expression evenly into each overlapping "bin".

```
<zenbu_script>

    <datastream name="gencode">

        <source name="UCSC_gencodeV10_hg19_20120101"  id="D71B7748-1450-4C62-92CB-7E913AB12899::19:::FeatureSource" category="gene"/>

    </datastream>

    <stream_processing>

        <spstream module="TemplateFilter">

            <overlap_mode>area</overlap_mode>
```

```
            <overlap_subfeatures>true</overlap_subfeatures>

            <side_stream>

                <spstream module="Proxy" name="gencode"/>

            </side_stream>

        </spstream>

        <spstream module="TemplateCluster">

            <overlap_subfeatures>true</overlap_subfeatures>

             <side_stream>

                <spstream module="FeatureEmitter">

                    <fixed_grid>true</fixed_grid>

                    <both_strands>true</both_strands>

                 </spstream>

            </side_stream>

        </spstream>

    </stream_processing>

</zenbu_script>
```

Example ZENBU view showing this script in use http:/ / fantom. gsc. riken. jp/ zenbu/ gLyphs/ #config=C5NmrYgRCxSkOQ0-St9HtC;loc=hg19::chr19:50351422..50392834

CutoffFilter

Data Stream Processing > Processing Modules > Filtering Modules

Description

The **CutoffFilter** processing module is designed to operate on the *significance* of Features and perform *high pass*, *low pass*, or *band pass* filtering. It is often used after **CalcFeatureSignificance** which combines the multiple experiment/expression data of a Feature into a single significance for that Feature.

Parameters

- **<min_cutoff>** : **Features** with significance less than min_cutoff are filtered out of the data stream.
- **<max_cutoff>** : **Features** with significance above max_cutoff are filtered out of the data stream.
- **<filter_by_experiment>** : set to *true* to perform filter testing at Experiment/Expression level (not feature significance) level. At least one Experiment in the collection of the Feature must pass the min/max filtering criteria. If no Experiment passes the criteria, the feature is removed from the data stream. The collection of Experiment/Expression is not altered.

Example

This is a complex script which incorporates a FeatureEmitter / TemplateCluster expression histogram binning with de-novo clustering via Paraclu followed by CalcFeatureSignificance and then several filtering steps including NeighborCutoff, **CutoffFilter**, and FeatureLengthFilter

```
<zenbu_script>
     <stream_queue>
          <spstream module="TemplateCluster">
               <overlap_mode>area</overlap_mode>
               <expression_mode>sum</expression_mode>
               <side_stream>
                    <spstream module="FeatureEmitter">
                         <width>1</width>
                         <fixed_grid>true</fixed_grid>
                         <both_strands>true</both_strands>
                    </spstream>
               </side_stream>
          </spstream>

          <spstream module="Paraclu">
               <min_cutoff>10</min_cutoff>
               <stability>0</stability>
               <max_cluster_length>100</max_cluster_length>
          </spstream>

          <spstream module="CalcFeatureSignificance">
               <expression_mode>sum</expression_mode>
          </spstream>
```
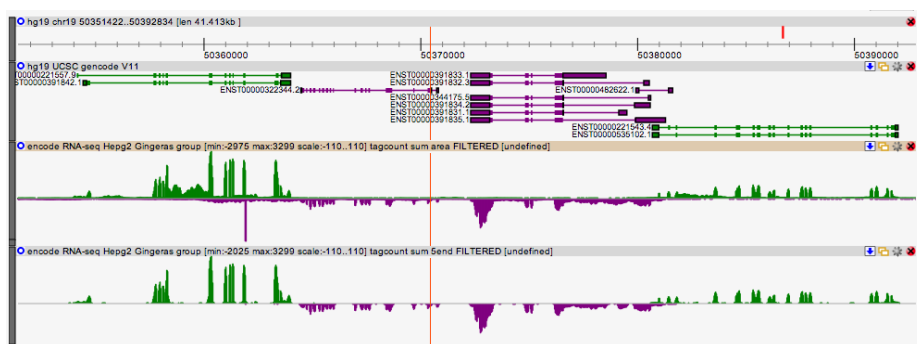
```
                <spstream module="NeighborCutoff">
                        <ratio>300</ratio>
                        <distance>100</distance>
                </spstream>

                <spstream module="CutoffFilter">
                        <min_cutoff>100</min_cutoff>
                </spstream>

                <spstream module="FeatureLengthFilter">
                        <max_length>50</max_length>
                </spstream>


        </stream_queue>
</zenbu_script>
```

Example ZENBU view showing this script in use

http://fantom. gsc. riken. jp/ zenbu/ gLyphs/ #config=W1Oe95W3Id5gZAhrlzIwlD;loc=hg19::chr19:49990236. . 49997699

ExpressionDatatypeFilter

Data Stream Processing > Processing Modules > Filtering Modules

Description

The **ExpressionDatatypeFilter** processing module is a simple filtering module which removes expression which does not match the specified datatypes. It operated on Features and removes Expression from them. It is often used in combination in complex scripts where multiple datatype are required at one stage of processing, but where the expression collection of the Features needs to be simplified at a later stage.

Parameters

• **<datatype>** : the datatype of expression which will be allowed to remain in the Features

Example

This script show a how ExpressionDatatypeFilter can be configured

```
<zenbu_script>
   <stream_queue>
      <spstream module="ExpressionDatatypeFilter">
              <datatype>tagcount_pm</datatype>
              <datatype>tpm</datatype>
      </spstream>
   </stream_queue>
</zenbu_script>
```

FeatureLengthFilter

Data Stream Processing > Processing Modules > Filtering Modules

Description

The **FeatureLengthFilter** processing module is designed to filter **Features** based on their length. The module is configurable with both a min_length and/or max_length. If only min_length is specified it will act as a high-pass filter. If only max_length is specified it will act as a low-pass filter. If both min_length and max_length are specified it will act as a band-pass filter.

Parameters

- **<min_length>** : Features smaller than min_length are filtered out.
- **<max_length>** : Features larger than max_length are filtered out.

Example

This is a complex script which incorporates a FeatureEmitter / TemplateCluster expression histogram binning with de-novo clustering via Paraclu followed by several filtering steps including NeighborCutoff, CutoffFilter, and a final **FeatureLengthFilter** to remove very tiny clusters.

```
<zenbu_script>
    <stream_queue>
        <spstream module="TemplateCluster">
            <overlap_mode>area</overlap_mode>
            <expression_mode>sum</expression_mode>
            <side_stream>
                <spstream module="FeatureEmitter">
                    <width>1</width>
                    <fixed_grid>true</fixed_grid>
                    <both_strands>true</both_strands>
                </spstream>
            </side_stream>
        </spstream>

        <spstream module="Paraclu">
            <min_cutoff>10</min_cutoff>
            <stability>0</stability>
            <max_cluster_length>100</max_cluster_length>
        </spstream>

        <spstream module="CalcFeatureSignificance"/>

        <spstream module="NeighborCutoff">
            <ratio>300</ratio>
            <distance>100</distance>
        </spstream>
```

```
                    <spstream module="CutoffFilter">
                            <min_cutoff>100</min_cutoff>
                    </spstream>

                    <spstream module="FeatureLengthFilter">
                            <max_length>50</max_length>
                    </spstream>


            </stream_queue>
</zenbu_script>
```

Example ZENBU view showing this script in use

http:/ / fantom. gsc. riken. jp/ zenbu/ gLyphs/ #config=jqr7D6J2PxMrOdqTg8glvD;loc=hg19::chr19:49990236. . 49997699

TopHits

Data Stream Processing > Processing Modules > Filtering Modules

Description

The TopHits processing module effectively returns a fixed number of Features in a stream query region based on a sorting of their significance. Since the total number of *top* Features is specified, a fixed queue length can be utilized and a running *top features* can be augmented in a streaming manner without requiring all data to be loaded into memory. After the top most significant Features have been found, they will be resorted based on chromosome location and streamed out. This module can be useful as a visualization filter to limit the number of objects on the screen, or can be useful for *'genome scanning* with data download to return only the *most significant* results.

Note: Currently the region of filtering is interacting with the TrackCache system so tophits are filtered on the fixed-gridding of the TrackCache system which is 100kb. If the track caching is disabled it will return the TopHits within the query region window. We will improve the control behavior of this module in the future.

Parameters

- **<queue_length>** : maximum number of *most significant* Features which will be returned
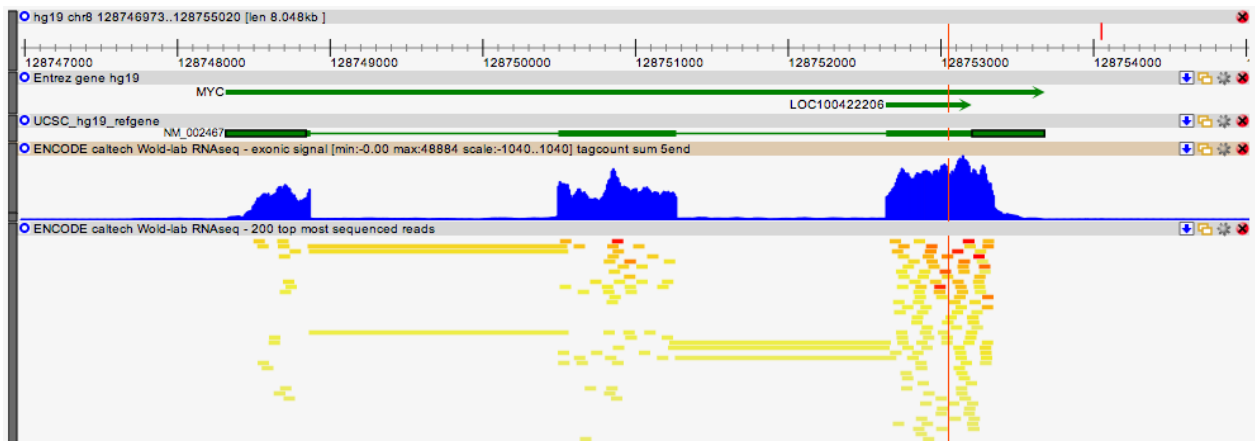
Example

This script show a how TopHits can be used as a visualization filter to make sure that the track does not explode with too many items and only shows the **most significant**.

```
<zenbu_script>
      <stream_queue>
            <spstream module="UniqueFeature">
                  <ignore_strand>true</ignore_strand>
            </spstream>
            <spstream module="TopHits">
                  <queue_length>200</queue_length>
            </spstream>
      </stream_queue>
</zenbu_script>
```

Example ZENBU view showing this script in use
http://fantom.gsc.riken.jp/zenbu/gLyphs/#config=FbcK_M26MlUnac19PXdOXB;loc=hg19::chr8:128746973..
128755020

NeighborCutoff

Data Stream Processing > Processing Modules > Filtering Modules

Description

The **NeighborCutoff** processing module is filtering algorithm which operates on the *significance* of Features. Filtering is performed based on the concept that "a hill next to a mountain is lost in the background, while that same hill in a field looks like a giant". Filtering is performed based on a ration of Features relative to their neighbors. Strong Features will **shadow** weaker Features and filter them out.

The motivation for NeighborCutoff was that often with sequence data, there are many situations where data appears to "spill over". When there is a strong signal, the background around that signal is often stronger than the background in other areas. Therefore it is sometime necessary to adjust the noise cutoff level relative to the signal in an area. This is what NeighborCutoff does.

Parameters

- **<distance>** : distance between Features which defines them to be *neighbors*
- **<ratio>** : maximum allow ratio of largest Feature in a Neighborhood to the smallest. Features less than (strongest neighbor significance / ratio) are filtered out. It can be consider like distance to the *noise floor*. The larger the *ratio* the more noisy / weaker neighbors are allowed to remain.

Example

This is a complex script which incorporates a FeatureEmitter / TemplateCluster expression histogram binning with de-novo clustering via Paraclu followed by CalcFeatureSignificance and then several filtering steps including NeighborCutoff, **CutoffFilter**, and FeatureLengthFilter

```
<zenbu_script>
    <stream_queue>
        <spstream module="TemplateCluster">
            <overlap_mode>area</overlap_mode>
            <expression_mode>sum</expression_mode>
            <side_stream>
                <spstream module="FeatureEmitter">
                    <width>1</width>
```

```
                            <fixed_grid>true</fixed_grid>
                            <both_strands>true</both_strands>
                    </spstream>
            </side_stream>
        </spstream>

        <spstream module="Paraclu">
                <min_cutoff>10</min_cutoff>
                <stability>0</stability>
                <max_cluster_length>100</max_cluster_length>
        </spstream>

        <spstream module="CalcFeatureSignificance">
                <expression_mode>sum</expression_mode>
        </spstream>

        <spstream module="NeighborCutoff">
                <ratio>300</ratio>
                <distance>100</distance>
        </spstream>

        <spstream module="CutoffFilter">
                <min_cutoff>100</min_cutoff>
        </spstream>

        <spstream module="FeatureLengthFilter">
                <max_length>50</max_length>
        </spstream>

    </stream_queue>
</zenbu_script>
```
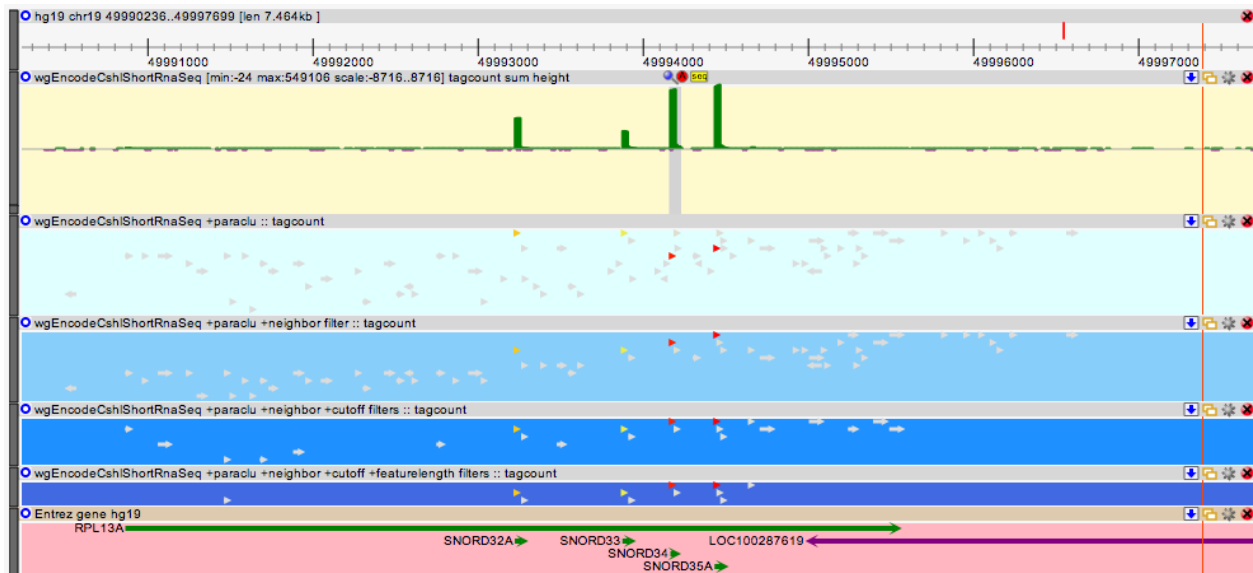
Example ZENBU view showing this script in use

http:/ / fantom. gsc. riken. jp/ zenbu/ gLyphs/ #config=jqr7D6J2PxMrOdqTg8glvD;loc=hg19::chr19:49990236. . 49997699

NormalizeByFactor

Data Stream Processing > Processing Modules > Data normalization and rescaling Modules

Description

The **NormalizeByFactor** processing module is designed to apply an Experiment specific normalization factor onto specified Expression data of streamed Features. It requires the Experiment specific normalization factor to be stored in metadata of the Experiment, and the specification of the input DataType and new datatype name.
The equation is simply

```
new_value = old_value * factor
```

Parameters

- **<experiment_metadata_tag>** : the metadata tag attached to the experiments which contains the normalization factor. This can be added to Experiments by using the Metadata editing interface or at data load time.
- **<datatype>** : the datatype of expression on the data stream which will be normalized. only one input <datatype> allowed.
- **<output_datatype>** : the datatype which the expression will be relabeled as

Example

This script show a simple situation where RLE normalization factor is applied to "tagcount" expression data and relabeled as "rle".

```
<zenbu_script>
   <stream_processing>
      <spstream module="NormalizeByFactor">
            <experiment_metadata_tag>RLE_normalization_factor</experiment_metadata_tag>
            <datatype>tagcount</datatype>
            <output_datatype>rle</output_datatype>
      </spstream>
   </stream_processing>
```

```
</zenbu_script>
```

NormalizePerMillion

Data Stream Processing > Processing Modules > Data normalization and rescaling Modules

Description

The **NormalizePerMillion** processing module is designed to apply the TPM (*tag-per-million*) normalization procedure onto expression data. When data is loaded into the ZENBU system from BED or OSCtable files, a **total** is calculated for every expression Experiment column. This total can be used to transform those expression values into a *per-million* normalized form.

Parameters

This module has no parameters

Example

Simple script...

```
<zenbu_script>
   <stream_queue>
      <spstream module="NormalizePerMillion"/>
   </stream_queue>
</zenbu_script>
```

NormalizeRPKM

Data Stream Processing > Processing Modules > Data normalization and rescaling Modules

Description

The **NormalizeRPKM** processing module is an extension of NormalizePerMillion and designed to normalize the expression of a feature based on both the total expression in an experiment and the cumulative length of each Feature's subfeatures to recompute the expression level as <datatype> per million per 1000 basepairs (RPKM). If there is no Experiment total count then normalization is just based on subfeature total length (<datatype> per 1000 basepairs).

Parameters

- **<category_filter>** : defines the subfeatures which are used in calculating the total subfeature length

Example

This is a script which incorporates a Proxy / TemplateCluster to collate expression into Gencode V10 gene models. The expression is then normalized via the NormalizeRPKM normalization module. The script finishes with CalcFeatureSignificance so that the Features can be displayed via score-coloring.

```
<zenbu_script>
    <datastream name="gencode" output="full_feature">
        <source id="D71B7748-1450-4C62-92CB-7E913AB12899::19:::FeatureSource"/>
    </datastream>
    <stream_queue>
        <spstream module="TemplateCluster">
            <overlap_mode>height</overlap_mode>
            <skip_empty_templates>false</skip_empty_templates>
            <expression_mode>sum</expression_mode>
            <overlap_subfeatures>true</overlap_subfeatures>
            <ignore_strand>true</ignore_strand>
            <side_stream>
                <spstream module="Proxy" name="gencode"/>
            </side_stream>
        </spstream>

        <spstream module="NormalizeRPKM">
            <category_filter>exon</category_filter>
            <category_filter>block</category_filter>
        </spstream>

        <spstream module="CalcFeatureSignificance"/>
    </stream_queue>
</zenbu_script>
```
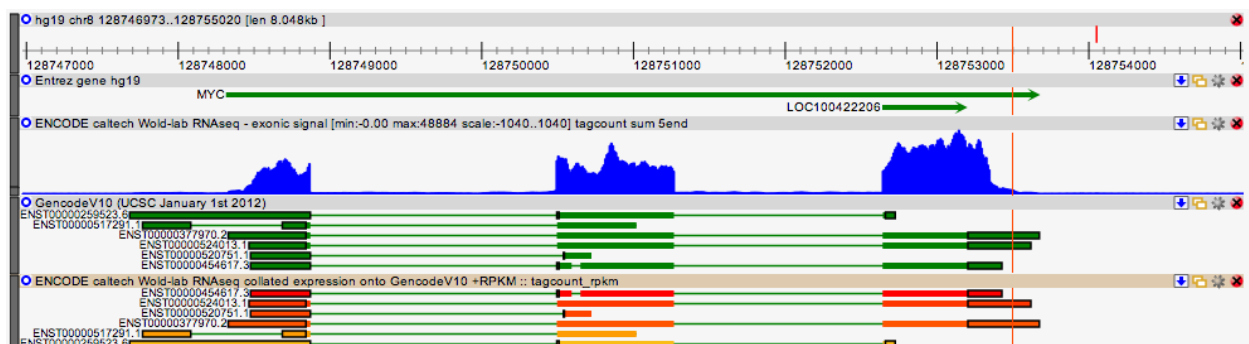
Here is a ZENBU view showing this script in use

http:/ / fantom. gsc. riken. jp/ zenbu/ gLyphs/

#config=vtPXLwwqO9KjD1YYWqCMGD;loc=hg19::chr8:128746973..128755020

RescalePseudoLog

Data Stream Processing > Processing Modules > Data normalization and rescaling Modules

Description

The **RescalePseudoLog** processing module is a simple signal processor which rescale expression level as pseudo log.

Log transformation is convenient to vizualize data whose expression levels varies in a wide range of values, but zero values are common places in genomic data and log(base,0) is not defined.

We would thus need to recurse to pseudocount (typically arbitrarily adding 1 or 0.5 to all values).

Alternatively we can use pseudolog defined as **asinh(x/2) / log(base)**, which has the following nice properties

- is defined for all real x values
- pseudolog(base, 0) = 0
- pseudolog(base, -x) = -1 * pseudolog(base, x)
- pseudolog(base, x) ~ log(base, x) for x > base

```
        For information :


            pseudolog(10,1)  = 0.2089876;    log10(1)  = 0
            pseudolog(10,2)  = 0.3827757;    log10(2)  = 0.3010300
            pseudolog(10,3)  = 0.5188791;    log10(3)  = 0.4771213
            pseudolog(10,4)  = 0.6269629;    log10(4)  = 0.6020600
            pseudolog(10,5)  = 0.7153834;    log10(5)  = 0.6989700
            pseudolog(10,10) = 1.0042792;    log10(10) = 1
            pseudolog(10,100)= 2.0000430;    log10(100)= 2


            pseudolog(2,1)   = 0.6942419;    log2(1)   = 0
            pseudolog(2,2)   = 1.2715533;    log2(2)   = 1
            pseudolog(2,3)   = 1.7236790;    log2(3)   = 1.584963
            pseudolog(2,4)   = 2.0827257;    log2(4)   = 2
            pseudolog(2,5)   = 2.3764522;    log2(5)   = 2.321928
            pseudolog(2,10)  = 3.3361433;    log2(10)  = 3.321928
            pseudolog(2,100) = 6.6440004;    log2(100) = 6.643856
```

Parameters

The sole optional parameter is the base of the pseudo log rescaling. By default, the rescaling base is set to 10

Example

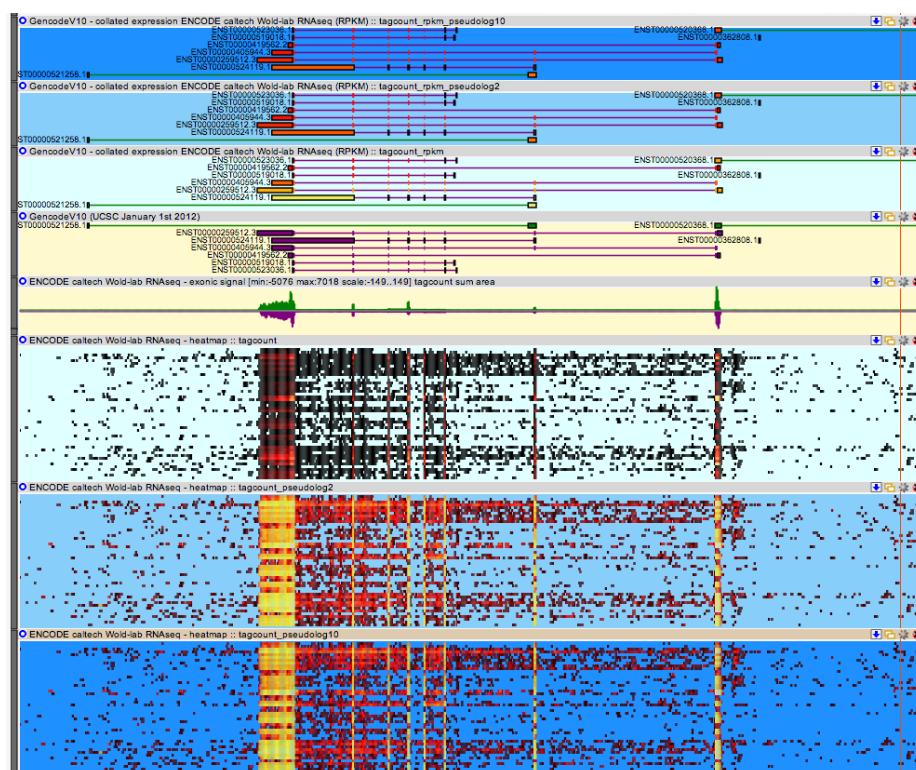Simple script...

```
<zenbu_script>
   <stream_queue>
      <spstream module="RescalePseudoLog">
          <base>10</base>
        </spstream>
   </stream_queue>
</zenbu_script>
```

An example of the usefulness of pseudolog rescaling is presented in the View "pseudolog rescaling example.1 " [1], with the wold lab ENCODE RNA-seq displayed as a heatmap and with expression collated and RPKM normalized onto gencodeV10 models and is rescaled as pseudo_log2 and pseudo_log10, providing a good overview of expression pattern dynamics throughout a wide range of expression levels.

http://fantom.gsc.riken.jp/zenbu/gLyphs/#config=ATMu21bIIZvgCegdIRpteD;loc=hg19::chr8:124010566..124067231

In the center of this view with a light yellow background RNAseq reads from the wold lab ENCODE dataset (unstranded protocol) and Gencode V10 transcript models.

On both sides, in increasingly darker shade of blue the RNAseq signal as a heatmap and as expression levels of gencode V10 transcripts (obtained by collating the RNAseq reads and normalized as RPKM), untouched, pseudo-log2, and pseudo-log20 rescaled.

References

[1] http://fantom.gsc.riken.jp/zenbu/gLyphs/#config=ATMu21bIIZvgCegdIRpteD;loc=hg19::chr8:124010566..124067231

CalcFeatureSignificance

Data Stream Processing > Processing Modules > General manipulation Modules

Description

The **CalcFeatureSignificance** processing module is designed to sit in the middle of a processing stream and transform the multiple Experiment / Expression data of a Feature into the single significance for that Feature.

Parameters

- **<expression_mode>** : defines how expression between different Experiments are combined together when calculating the Feature significance. Possible values are:

  - **sum** : sum the expression between different Experiments into the significance.
  - **min** : calculate the minimum expression value among different Experiments
  - **max** : calculate the maximum expression value among different Experiments
  - **count** : count the number of different Experiments of the Feature.
  - **mean** : calculate the average expression value among the different Experiments of the feature

Example

This script combines **FeatureEmitter / TemplateCluster** strandless, expression histogram binning with a **CalcFeatureSignificance**. This can then be visualized in a hybrid track using a color spectrum.

```
<zenbu_script>
   <parameters>
      <source_outmode>skip_metadata</source_outmode>
      <skip_default_expression_binning>true</skip_default_expression_binning>
   </parameters>
   <stream_stack>
      <spstream module="CalcFeatureSignificance">
           <expression_mode>sum</expression_mode>
      </spstream>
      <spstream module="TemplateCluster">
           <overlap_mode>height</overlap_mode>
           <expression_mode>sum</expression_mode>
           <ignore_strand>true</ignore_strand>
           <overlap_subfeatures>true</overlap_subfeatures>
           <side_stream>
                <spstream module="FeatureEmitter">
                     <num_per_region>970</num_per_region>
                     <fixed_grid>true</fixed_grid>
                     <both_strands>false</both_strands>
                </spstream>
           </side_stream>
```

```
    </spstream>
</zenbu_script>
```

CalcInterSubfeatures

Data Stream Processing > Processing Modules > General manipulation Modules

Description

The **CalcInterSubfeatures** processing module is designed to work on Features with subfeatures and *fill in the gaps* between subfeatures with new subfeatures. The subfeatures used in the calculation can be filtered and the *category* of the newly created subfeatures is named.

Parameters

- **<category_filter>** : defines the subfeatures which are used to demarcate the areas where the new subfeatures are created between.
- **<new_subfeature_category>** : defines the category name of the new FeatureSource for the newly created *inter* subfeatures

Example

This script combines two CalcInterSubfeatures modules with a StreamSubfeatures module to manipulate a complex gene model of exons and UTRs into an exported set of coding exons. It first creates *intron* features between the exons which can be labeled either as *block* or *exon*. The second CalcInterSubfeatures then uses *5utr* and *3utr* and *intron* as demarcation to create *codingexon* subfeatures. Lastly the StreamSubfeatures exports these *codingexon* subfeatures out onto the primary data stream.

```
<zenbu_script>
  <stream_queue>
        <spstream module="CalcInterSubfeatures">
              <category_filter>block</category_filter>
              <category_filter>exon</category_filter>
              <new_subfeature_category>intron</new_subfeature_category>
        </spstream>
        <spstream module="CalcInterSubfeatures">
              <category_filter>5utr</category_filter>
              <category_filter>3utr</category_filter>
              <category_filter>intron</category_filter>
              <new_subfeature_category>codingexon</new_subfeature_category>
        </spstream>
        <spstream module="StreamSubfeatures">
              <category_filter>codingexon</category_filter>
              <transfer_expression>true</transfer_expression>
        </spstream>
  </stream_queue>
</zenbu_script>
```

Here is a ZENBU view showing this script in use

http://fantom. gsc. riken. jp/ zenbu/ gLyphs/ #config=Z-bDHnDrB9UiZmORbpzGl;loc=hg19::chr8:128746973. . 128755020

StreamSubfeatures

Data Stream Processing > Processing Modules > General manipulation Modules

Description

The **SteramSubfeatures** processing module is designed to work on Features with subfeatures and exports them onto the primary data stream. The subfeatures can be selected based on their *category*. In addition there are parameter control options related to expression and uniquing.

Parameters

- **<category_filter>** : defines the subfeatures which are exported from the primary feature.
- **<transfer_expression>** : if *true* the expression from the primary Feature is transfered to all the subfeatures equally using the *height* expression method described in TemplateCluster.
- **<unique>** : internall implements the same algorithm as the UniqueFeature module.

Example

This script combines two CalcInterSubfeatures modules with a StreamSubfeatures module to manipulate a complex gene model of exons and UTRs into an exported set of coding exons. It first creates *intron* features between the exons which can be labeled either as *block* or *exon*. The second CalcInterSubfeatures then uses *5utr* and *3utr* and *intron* as demarkation to create *codingexon* subfeatures. Lastly the StreamSubfeatures exports these *codingexon* subfeatures out onto the primary data stream.

```
<zenbu_script>
  <stream_queue>
        <spstream module="CalcInterSubfeatures">
              <category_filter>block</category_filter>
              <category_filter>exon</category_filter>
              <new_subfeature_category>intron</new_subfeature_category>
        </spstream>
        <spstream module="CalcInterSubfeatures">
              <category_filter>5utr</category_filter>
              <category_filter>3utr</category_filter>
              <category_filter>intron</category_filter>
              <new_subfeature_category>codingexon</new_subfeature_category>
        </spstream>
        <spstream module="StreamSubfeatures">
              <category_filter>codingexon</category_filter>
              <transfer_expression>true</transfer_expression>
        </spstream>
  </stream_queue>
</zenbu_script>
```

Here is a ZENBU view showing this script in use

http://fantom. gsc. riken. jp/ zenbu/ gLyphs/ #config=Z-bDHnDrB9UiZmORbpzGl;loc=hg19::chr8:128746973. .
128755020

FilterSubfeatures

Data Stream Processing > Processing Modules > General manipulation Modules

Description

The **FilterSubfeatures** processing module is designed to work on Features with subfeatures to remove subfeatures not specified in the category filtering and then rebuild the outer feature boundaries based on the remaining subfeatures. The module will resort features on the data stream as needed to preserve the stream integrity.

Parameters

- **<category_filter>** : defines the subfeatures which are retained in the rebuilt primary feature.

Example

This script combines two CalcInterSubfeatures modules with a **FilterSubfeatures** module to manipulate a complex gene model of exons and UTRs into a new gene model based on coding exons. It first creates *intron* features between the exons which might be labeled either as *block* or *exon*. The second CalcInterSubfeatures then uses *5utr* and *3utr* and *intron* as demarkation to create *codingexon* subfeatures. Lastly the **FilterSubfeatures** rebuilds the transcript gene model using only these *codingexon* subfeatures.

```
<zenbu_script>
  <stream_queue>
        <spstream module="CalcInterSubfeatures">
              <category_filter>block</category_filter>
              <category_filter>exon</category_filter>
              <new_subfeature_category>intron</new_subfeature_category>
        </spstream>
        <spstream module="CalcInterSubfeatures">
              <category_filter>5utr</category_filter>
              <category_filter>3utr</category_filter>
              <category_filter>intron</category_filter>
              <new_subfeature_category>codingexon</new_subfeature_category>
        </spstream>
        <spstream module="FilterSubfeatures">
              <category_filter>codingexon</category_filter>
        </spstream>
    </stream_queue>
</zenbu_script>
```

Here is a ZENBU view showing this script in use

http://fantom.gsc.riken.jp/zenbu/gLyphs/#config=adzrWGcdF1GoZMET-n_LRC;loc=hg19::chr8:128746973..
128755020

ResizeFeatures

Data Stream Processing > Processing Modules > General manipulation Modules

Description

The **ResizeFeatures** processing module is designed to work on Features to alter their genomic coordinates. The module will resort features on the data stream as needed to preserve the stream integrity. Typical use cases

- shrink the feature to its 5' end and make it 1bp wide (CAGE data)
- expand a feature 500bp upstream of it's 5'end (ex extending refseq genes into their potential promoter regions to use for overlap analysis)

Parameters

- **<category_filter>** : only Features matching categories will be resized
- **<mode>** : defines which resize method will be performed

  - **shrink_start** : shrink the feature to 1bp length on the chrom_start
  - **shrink_end** : shrink the feature to 1bp length on the chrom_end
  - **shrink_5prime** : shrink the feature to 1bp length on the 5' end (start for + strand, end of - strand)
  - **shrink_3prime** : shrink the feature to 1bp length on the 3' end (start for - strand, end of + strand)
  - **expand_start** : expand the chrom_start by <expand> amount up-stream
  - **expand_end** : expand the chrom_end by <expand> amount down-stream
  - **expand_5prime** : expand the 5' end by <expand> amount
  - **expand_3prime** : expand the 3' end by <expand> amount
  - **store** : save the coordinates prior to resizing, such that they can be restored in a latter call to ResizeFeatures
  - **restore** : redefine the coordinates as those stored in a previous ResizeFeatures call
- **<expand>** : for *expand* modes this is the amount to be expanded.
- **<retain_subfeatures>** : Defaults to "false" which is the desired behavior: after resising the subfeatures have no more meaning, unless the original coordinates are thought to be store(d)/restore(d) back. Hence, this flag must be set to "true" if you plan to redefine the coordinates to those stored in a previous ResizeFeatures call along with its subfeatures.
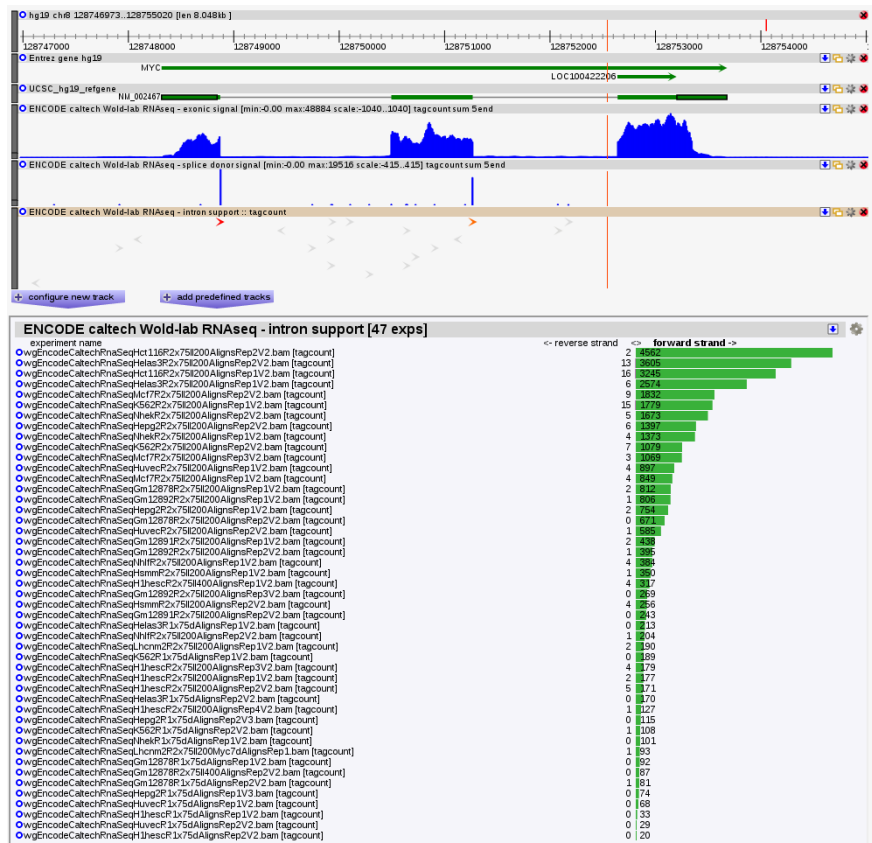
Example

This script combines a CalcInterSubfeatures modules with a StreamSubfeatures module to generate *introns*. This is followed by ResizeFeatures and UniqueFeature to reduce the introns into a set of unique intron donor sites with counts of their abundance.

```
<zenbu_script>
    <stream_processing>
        <spstream module="CalcInterSubfeatures"/>
        <spstream module="StreamSubfeatures">
            <category_filter>intron</category_filter>
            <transfer_expression>true</transfer_expression>
            <unique>
                <ignore_strand>true</ignore_strand>
            </unique>
        </spstream>
```

```
                <spstream module="ResizeFeatures">
                        <mode>shrink_start</mode>
                </spstream>
                <spstream module="UniqueFeature"/>
        </stream_processing>
</zenbu_script>
```

Here is a ZENBU view showing this script in use

http://fantom.gsc.riken.jp/zenbu/gLyphs/#config=fuf2V6ehKhHlbabZkQWebB;loc=hg19::chr8:128746973..
128755020

This script exemplifies how one would collate CAGE based expression into transcritps. It uses the combination of several consecutive ResizeFeatures modules, the first ones defining the [-500..+500] region around the beginning of transcritps from which CAGE TSS can be associated (using the with a TemplateCluster module) to the transcripts, and a last final one restoring the original transcript coordinates and their associated exons structure (subfeatures)

```
<zenbu_script>

    <note> Lists RefSeq transcript sources for all the main assemblies loaded in zenbu </note>

    <datastream name="refseq" output="full_feature">

        <source id="025F0224-D145-4E28-86A2-DB37A42A89CB::21:::FeatureSource" name="UCSC_RefSeq_canFam2_20120101"/>

        <source id="025F0224-D145-4E28-86A2-DB37A42A89CB::5:::FeatureSource" name="UCSC_RefSeq_galGal3_20120101"/>

        <source id="D71B7748-1450-4C62-92CB-7E913AB12899::13:::FeatureSource" name="UCSC_RefSeq_hg19_20120101"/>

        <source id="4043B030-0201-495F-824B-BC197EA3C272::6:::FeatureSource" name="UCSC_RefSeq_mm9_20120101"/>

        <source id="025F0224-D145-4E28-86A2-DB37A42A89CB::35:::FeatureSource" name="UCSC_RefSeq_rn4_20120101"/>

        <source id="0583D02E-BA10-11DE-B45C-8D369A8382FD::78:::FeatureSource" name="UCSC_hg18_refgene"/>

    </datastream>


    <stream_processing>
```

```
        <note> Get the Transcriptional Start Sites (TSS) revealed by the 5'extremity of CAGE derived reads </note>

<spstream module="ResizeFeatures">

            <mode>shrink_5prime</mode>

    </spstream>


    <note> Collate the CAGE TSS along regions defined as RefSeq TSS +/-500bp </note>

    <spstream module="TemplateCluster">

            <ignore_strand value="false"/>

            <side_stream>

                <spstream module="Proxy" name="refseq"/>


                <note> Modify the coordinates to refseq TSS, save temporaly the original coordinates for later call back </note>

                <spstream module="ResizeFeatures">

                    <retain_subfeatures>true</retain_subfeatures>

                    <mode>store</mode>

                </spstream>


                <note> Modify the coordinates to refseq TSS+/-500bp </note>

                <spstream module="ResizeFeatures">

                    <retain_subfeatures>true</retain_subfeatures>

                    <mode>shrink_5prime</mode>

                </spstream>

                <spstream module="ResizeFeatures">

                    <retain_subfeatures>true</retain_subfeatures>

                    <mode>expand_start</mode>

                    <expand>500</expand>

                </spstream>

                <spstream module="ResizeFeatures">

                    <retain_subfeatures>true</retain_subfeatures>

                    <mode>expand_end</mode>

                    <expand>500</expand>

                </spstream>


            </side_stream>

    </spstream>


    <note> Restore RefSeq original coordinates </note>

    <spstream module="ResizeFeatures">

            <retain_subfeatures>true</retain_subfeatures>

            <mode>restore</mode>

    </spstream>


    <note> Sum up the expression over all samples and save the value as the refseq score to color it accordingly </note>

   <spstream module="CalcFeatureSignificance">

    <expression_mode>sum</expression_mode>

    </spstream>
```
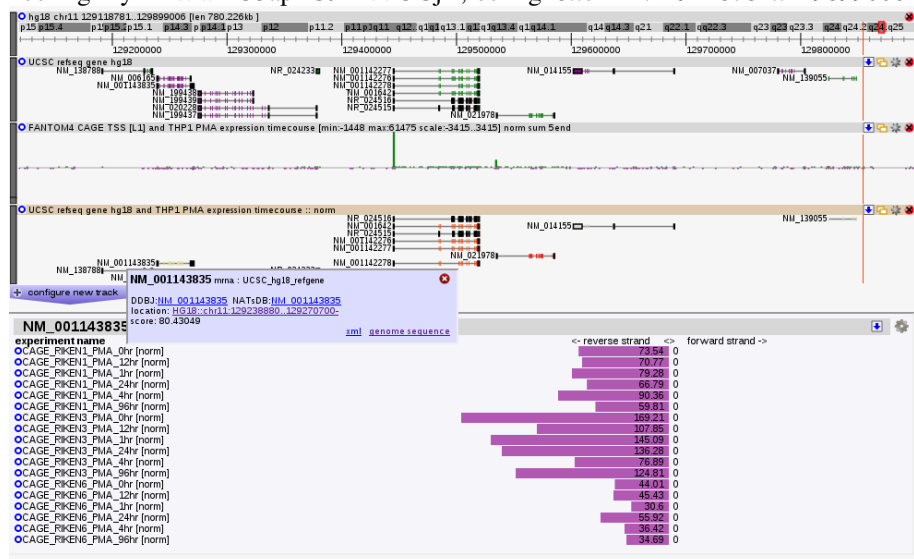
```
    </stream_processing>

</zenbu_script>
```

Here is a ZENBU view showing this script in use

http:/ / fantom. gsc. riken. jp/ zenbu/ gLyphs/ #config=PyTxIWwAO5apFGJYNVOGjB;loc=hg18::chr11:129118781..129899006

MakeStrandless

Data Stream Processing > Processing Modules > General manipulation Modules

Description

The **MakeStrandless** processing module processes **Features** on the data stream and removes their strand information to make them strandlass. Useful when the original data might be alignments which shows alignment strand, but the experimental sample protocol is actually strandless in nature and thus the strand in the alignment is meaningless.

Parameters

This module has no parameters

Example

Simple script...

```
<zenbu_script>
    <stream_queue>
        <spstream module="MakeStrandless"/>

        <spstream module="TemplateCluster">
            <overlap_mode>height</overlap_mode>
            <expression_mode>sum</expression_mode>
            <overlap_subfeatures>true</overlap_subfeatures>
```

```
                        <ignore_strand>true</ignore_strand>
                        <side_stream>
                              <spstream module="FeatureEmitter">
                                    <num_per_region>970</num_per_region>
                                    <fixed_grid>true</fixed_grid>
                                    <both_strands>false</both_strands>
                              </spstream>
                        </side_stream>
                  </spstream>
            </stream_stack>
</zenbu_script>
```

RenameExperiments

Data Stream Processing > Processing Modules > General manipulation Modules

Description

The **RenameExperiments** processing module processes **Experiment** and changes their name to a concatenation of metadata identified by their metadata tag.

Parameters

• **<tag prefix="''">** : search Experiment for metadata with this 'tag', with option *prefix* and append metadata *value* as new experiment name. Specify multiple <tag> entries to concatenate new name together from multiple metadata elements.

Example

Simple script...

```
<zenbu_script>
      <stream_processing>
            <spstream module="RenameExperiments">
                  <tag>enc:cell</tag>
                  <tag>enc:cell_lineage</tag>
                  <tag>enc:cell_tissue</tag>
            </spstream>
      </stream_processing>
</zenbu_script>
```

Example ZENBU view showing this script in use
http://fantom.gsc.riken.jp/zenbu/gLyphs/#config=1sN5FmqjDRrhN2yia0d5SB;loc=hg19::chr8:128746973..
128755020

FeatureRename

Data Stream Processing > Processing Modules > General manipulation Modules

Description

The **FeatureRename** processing module processes **Features** and changes their name. Current version sets the name of the Feature to be the name of the Feature's **FeatureSource**

Parameters

This module has no parameters

Example

Simple script...

```
<zenbu_script>
      <stream_processing>
            <spstream module="FeatureRename"/>
      </stream_processing>
</zenbu_script>
```

Article Sources and Contributors

Data Stream Processing *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=3098 *Contributors*: Jessica Severin, Nicolas.bertin

Data Sources *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2039 *Contributors*: Jessica Severin, Nicolas.bertin

Data Stream Pool *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2740 *Contributors*: Jessica Severin

Data Abstraction Model *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2744 *Contributors*: Jessica Severin, Nicolas.bertin

TrackCaching System *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2638 *Contributors*: Nicolas.bertin

Proxy *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=3267 *Contributors*: Jessica Severin, Nicolas.bertin

FeatureEmitter *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=3061 *Contributors*: Jessica Severin, Nicolas.bertin

TemplateCluster *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=3253 *Contributors*: Jessica Severin, Nicolas.bertin

UniqueFeature *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2407 *Contributors*: Jessica Severin, Nicolas.bertin

Paraclu *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2417 *Contributors*: Jessica Severin, Nicolas.bertin

TemplateFilter *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=3245 *Contributors*: Jessica Severin, Nicolas.bertin

CutoffFilter *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2511 *Contributors*: Jessica Severin, Nicolas.bertin

ExpressionDatatypeFilter *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2512 *Contributors*: Jessica Severin, Nicolas.bertin

FeatureLengthFilter *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2418 *Contributors*: Jessica Severin, Nicolas.bertin

TopHits *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2519 *Contributors*: Jessica Severin, Nicolas.bertin

NeighborCutoff *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2423 *Contributors*: Jessica Severin, Nicolas.bertin

NormalizeByFactor *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2425 *Contributors*: Jessica Severin, Nicolas.bertin

NormalizePerMillion *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2426 *Contributors*: Jessica Severin, Nicolas.bertin

NormalizeRPKM *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2441 *Contributors*: Jessica Severin, Nicolas.bertin

RescalePseudoLog *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2443 *Contributors*: Nicolas.bertin

CalcFeatureSignificance *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2432 *Contributors*: Jessica Severin, Nicolas.bertin

CalcInterSubfeatures *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2433 *Contributors*: Jessica Severin, Nicolas.bertin

StreamSubfeatures *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2431 *Contributors*: Jessica Severin, Nicolas.bertin

FilterSubfeatures *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2434 *Contributors*: Jessica Severin, Nicolas.bertin

ResizeFeatures *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2435 *Contributors*: Jessica Severin, Nicolas.bertin

MakeStrandless *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2436 *Contributors*: Jessica Severin, Nicolas.bertin

RenameExperiments *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2440 *Contributors*: Jessica Severin, Nicolas.bertin

FeatureRename *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?oldid=2439 *Contributors*: Jessica Severin, Nicolas.bertin

Image Sources, Licenses and Contributors

File:Configure_custom_XML_script.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Configure_custom_XML_script.png *License*: unknown *Contributors*: Jessica Severin

File:Data-pooling-repeat-example.jpg *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Data-pooling-repeat-example.jpg *License*: unknown *Contributors*: Jessica Severin

File:Data-pooling-experiment-example.jpg *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Data-pooling-experiment-example.jpg *License*: unknown *Contributors*: Jessica Severin

File:EEDB_DataModel.jpg *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:EEDB_DataModel.jpg *License*: unknown *Contributors*: Jessica Severin

File:Gencode_collation_RNAseq.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Gencode_collation_RNAseq.png *License*: unknown *Contributors*: Jessica Severin

File:Track_controls-reconfigure_track.jpg *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Track_controls-reconfigure_track.jpg *License*: unknown *Contributors*: Jessica Severin

File:Datastream_xml_widget.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Datastream_xml_widget.png *License*: unknown *Contributors*: Jessica Severin

Image:FeatureEmitter.1.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:FeatureEmitter.1.png *License*: unknown *Contributors*: Nicolas.bertin

Image:paraclu.1.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Paraclu.1.png *License*: unknown *Contributors*: Nicolas.bertin

Image:templatefilter.1.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Templatefilter.1.png *License*: unknown *Contributors*: Nicolas.bertin

Image:tophit.1.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Tophit.1.png *License*: unknown *Contributors*: Nicolas.bertin

Image:Gencode.rpkm.collation.1.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Gencode.rpkm.collation.1.png *License*: unknown *Contributors*: Nicolas.bertin

image:pseudolog_rescaling.1.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:Pseudolog_rescaling.1.png *License*: unknown *Contributors*: Nicolas.bertin

File:ProcessingResizeFeatures.SpliceDonorAcceptor.example.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:ProcessingResizeFeatures.SpliceDonorAcceptor.example.png *License*: unknown *Contributors*: Nicolas.bertin

File:ProcessingResizeFeatures.RefseqPromCAGEexpression.example.png *Source*: http://fantom.gsc.riken.jp/zenbu/wiki/index.php?title=File:ProcessingResizeFeatures.RefseqPromCAGEexpression.example.png *License*: unknown *Contributors*: Nicolas.bertin

License