Drop-seq Core Computational Protocol

version 2.0.0 (9/28/18)

James Nemesh
Steve McCarroll's lab, Harvard Medical School

Introduction

The following is a manual for using the software we have written for processing Drop-seq sequence data into a "digital expression matrix" that will contain integer counts of the number of transcripts for each gene, in each cell. This software pipeline performs many analyses including massive de-multiplexing of the data, alignment of reads to a reference genome, and processing of cellular and molecular barcodes.

Drop-seq sequencing libraries produce paired-end reads: read 1 contains both a cell barcode and a molecular barcode (also known as a UMI); read 2 is aligned to the reference genome. This document provides step-by-step instructions for using the software we have developed to convert these sequencing reads into a digital expression matrix that contains integer counts of the number of transcripts for each gene, in each cell.

We may release updates to this manual as we learn from users' experiences. If a revision simply contains additional hints or advice or detail, then we will update the date on the protocol but not the version number. Whenever we implement a substantive change to the software or protocol, we will increment the version number.

We hope this is helpful and that you are soon generating exciting data with Drop-seq.

Introduction V2:
There are a number of enhancements to the Drop-seq platform that come with version 2.0: new methods to clean up the cell barcodes from bead synthesis errors and PCR errors result in less clutter in the data when trying to decide which cell barcodes are truly cells. We've also enhanced Digital expression to be more flexible in how it interprets gene annotations, allowing the program to extract both intronic DGE data as well as the typical coding+utr data. Read on to find out about new Drop-seq program capabilities in-line with the rest of the documentation.

Drop-seq Software and Hardware Requirements

The Drop-seq software provided is implemented entirely in Java. This means it will run on a huge number of devices that are capable of running Java, from large servers to laptops. We require 4 gigabytes of memory for each program to run, which is also sufficient for Picard programs we use as part of alignment and analysis. Disk space will be determined by your data size plus the meta-data and aligner index. 50 gigabytes of disk space will be sufficient to store our meta data plus a STAR index.

Overview of Alignment

The raw reads from the sequencer must be converted into a Picard-queryname-sorted BAM file for each library in the sequencer run. Since there are many sequencers and pipelines available to do this, we leave this step to the user. For example, we use either Picard IlluminaBasecallsToSam (preceded by Picard ExtractIlluminaBarcodes for a library with sample barcodes); or Illumina's bcl2fastq followed by Picard FastqToSam**.** Once you have an unmapped, queryname-sorted BAM, you can follow this set of steps to align your raw reads and create a BAM file that is suitable to produce digital gene expression (DGE) results.

1. Unmapped BAM -> aligned and tagged BAM
    a. Tag cell barcodes
    b. Tag molecular barcodes
    c. Trim 5' primer sequence
    d. Trim 3' polyA sequence
    e. SAM -> Fastq
    f. STAR alignment
    g. Sort STAR alignment in queryname order
    h. Merge STAR alignment tagged SAM to recover cell/molecular barcodes
    i. Add gene/exon and other annotation tags
    j. Barcode Repair
        i. Repair substitution errors (DetectBeadSubstitutionErrors)
        ii. Repair indel errors (DetectBeadSynthesisErrors)

A walkthrough of the alignment process

Let's walk through these steps to help you build intuition about how reads are manipulated - later parts of this document will detail the software and invocations necessary to carry out these operations. First, you'll take your Drop-seq experiment and put it on a sequencer. The sequencer will gather data from both reads of the read pair. Read one is a *barcoded read,* containing the cell and molecular barcodes that will later identify this read as coming from a particular transcript on a particular cell. Read two is the *biological read*, which contains a portion of the sequence of the transcript observed.

First, you'll make a BAM file out of this data so that these two reads are in the same place. Then, we'll transfer information from the barcoded read over to the BAM record containing the genome read as a set of BAM tags. The first 12 bases of the barcoded read contain the cell barcode, so we'll copy those bases over to a BAM tag (XC) on the genome read. Then we'll take the next 8 bases containing the molecular barcode and copy them over as another BAM tag (XM). Since we're now extracted all the information out of the barcoded read, we discard the read, converting the BAM to single-ended reads.

If a barcoded read has low quality base, both the barcoded read and genome read are purged at this point. This makes life a lot easier for us in the future, as we don't have to track the barcoded read to know the origins of any genome read.

After this, we clean up the genome read with a few processes. The 5' adapter is detected and trimmed, as are 3' poly A tails. We call this final cleaned-up BAM the unaligned BAM. Then, we want to align these single-ended genome reads to the genome using STAR. To do this, we extract the fastq file containing single-ended reads from our genome read BAM file and run STAR. After STAR is done aligning reads, we now know where the genome reads align, but we've lost track of what cell and molecular barcodes these reads have. This information is recovered by merging the BAM tags from the unaligned BAM to the aligned reads from STAR. We then add additional annotation to the reads that is dependent on the genome read, such as any genes or exons that the read overlaps. Finally, we check for bead synthesis errors and repair them if possible.

The next sections will explain the metadata needed to follow this workflow, as well as explain each of the programs that have been developed to run these steps. Some of these programs are developed by us, and others take advantage of existing Picard Tools or aligners like STAR.

Metadata

To follow this set of processes from raw unaligned reads to an aligned BAM, it's necessary to have a number of different metadata files. These provide information about the sequence of the organism(s) you're running your experiment on, as well as genomic features like genes,transcripts, and exons that help extract DGE data from the reads.

We organize our metadata using a set of conventions we suggest you follow, as it makes it easier to keep track of what files are used for particular processes. In the software section, we'll refer to these files using these conventions.

The first convention is that we establish a root name for all of our files that encodes information about the organism and the genome build used to derive that metadata. For example, mm10 is the Dec. 2011 *Mus musculus* assembly. All files for mouse use this as the root name, followed by a ".", then the type of file.

metadata file types:

- *fasta*: The reference sequence of the organism. Needed for most aligners.
- *dict*: A dictionary file as generated by Picard's CreateSequenceDictionary. Needed for Picard Tools.

- gtf: The principle file to determine the location of genomic features like genes, transcripts, and exons. Many other metadata files we use derive from this original file. We download our GTF files from ensembl, which has a handy description of the file format here. Ensembl has a huge number of prepared GTF files for a variety of organisms here.
- refFlat: This file contains a subset of the the same information in the GTF file in a different format. Picard tools like the refFlat format, so we require this as well. To make life easy, we provide a program ConvertToRefFlat that can convert files from GTF format to refFlat for you.
- genes.intervals: The genes from the GTF file in interval list format. This file is optional, and useful if you want to go back to your BAM later to see what gene(s) a read aligns to.
- exons.intervals: The exons from the GTF file in interval list format. This file is optional, and useful if you want to go back to your BAM and view what exon(s) a read aligns to.
- rRNA.intervals: The locations of ribosomal RNA in interval list format. This file is optional, but we find it useful to later assess how much of a dropseq library aligns to rRNA.
- reduced.gtf: This file contains a subset of the information in the GTF file, but in a far more human readable format. This file is optional, but can be generated easily by the supplied ReduceGTF program that will take a GTF file as input.

On the Drop-Seq website you will find a set of pre-made meta data for human, mouse and human/mouse experiments.

Premade Meta Data links @GEO.
MIXED MOUSE HUMAN

MetaData Creation Programs
A few files and required to generate meta data: a GTF file, and a fastq file. From these two files we can derive various other files needed by the the Drop-seq software.

CreateSequenceDictionary
The first file needed is the sequence dictionary. This is a list of the contigs in the fastq file and their lengths.

```
java -jar /path/to/picard/picard.jar CreateSequenceDictionary
REFERENCE=my.fasta
OUTPUT= my.dict
SPECIES=species_name
```

ConvertToRefFlat
The next file is the refFlat file, which is generated using the sequence dictionary generated above.

```
ConvertToRefFlat
ANNOTATIONS_FILE=my.gtf
SEQUENCE_DICTIONARY=my.dict
OUTPUT=my.refFlat
```

ReduceGTF
The may be useful if you need an easy to parse version of your annotations in a language like R, and is also used to generate the other metadata.

ReduceGTF
SEQUENCE_DICTIONARY=my.dict
GTF=my.gtf
OUTPUT=my.reduced.gtf

CreateIntervalsFiles
As a last step, we create interval files needed for various programs in the Drop-seq pipeline. This program generates a number of interval files for genes, exons, consensus introns, rRNA, and mt. The example below uses the human MT contig name, but if you use a different organism you should set that argument appropriately.

CreateIntervalsFiles
SEQUENCE_DICTIONARY=my.dict
REDUCED_GTF=my.reduced.gtf
PREFIX=my
OUTPUT=/path/to/output/files
MT_SEQUENCE=MT

MetaData Generation Pipeline

We've provided a shell script to generate new meta data sets for single organism data in the distribution. This script is called create_Drop-seq_reference_metadata.sh, and the options for the program can be accessed by running with the -h option:
/path/to/dropseq_tools/create_Drop-seq_reference_metadata.sh -h

Alignment Pipeline Programs

On the Drop-seq website you will find a zipfile containing the programs described below. The zipfile also contains a script Drop-seq_alignment.sh that executes the process described below. Because of differences in computing environments, this script is not guaranteed to work for all users. However, we hope it will serve as an example of how the various programs should be invoked.

TagBamWithReadSequenceExtended
This Drop-seq program extracts bases from the cell/molecular barcode encoding read (BARCODED_READ), and creates a new BAM tag with those bases on the *genome read*. By default, we use the BAM tag XM for molecular barcodes, and XC for cell barcodes, using the TAG_NAME parameter.

This program is run once per barcode extraction to add a tag. On the first iteration, the cell barcode is extracted from bases 1-12. This is controlled by the BASE_RANGE option. On the second iteration, the molecular barcode is extracted from bases 13-20 of the barcode read. This program has an option to drop a read (DISCARD_READ), which we use after both barcodes have been extracted, which makes the output BAM have unpaired reads with additional tags.

Additionally, this program has a BASE_QUALITY option, which is the minimum base quality of all bases of the barcode being extracted. If more than NUM_BASES_BELOW_QUALITY bases falls below this quality, the read pair is discarded.

Example Cell Barcode:
TagBamWithReadSequenceExtended
INPUT=my_unaligned_data.bam
OUTPUT=unaligned_tagged_Cell.bam
SUMMARY=unaligned_tagged_Cellular.bam_summary.txt
BASE_RANGE=1-12
BASE_QUALITY=10
BARCODED_READ=1
DISCARD_READ=False
TAG_NAME=XC
NUM_BASES_BELOW_QUALITY=1

Example Molecular Barcode:
TagBamWithReadSequenceExtended
INPUT=unaligned_tagged_Cell.bam
OUTPUT=unaligned_tagged_CellMolecular.bam
SUMMARY=unaligned_tagged_Molecular.bam_summary.txt
BASE_RANGE=13-20
BASE_QUALITY=10
BARCODED_READ=1
DISCARD_READ=True
TAG_NAME=XM
NUM_BASES_BELOW_QUALITY=1

FilterBam:
This Drop-seq program is used to remove reads where the cell or molecular barcode has low quality bases. During the run of TagBamWithReadSequenceExtended, an XQ tag is added to each read to represent the number of bases that have quality scores below the BASE_QUALITY threshold. These reads are then removed from the BAM.

Example:
FilterBam
TAG_REJECT=XQ

INPUT=unaligned_tagged_CellMolecular.bam
OUTPUT=unaligned_tagged_filtered.bam

TrimStartingSequence

This Drop-seq program is one of two sequence cleanup programs designed to trim away any extra sequence that might have snuck it's way into the reads. In this case, we trim the SMART Adapter that can occur 5' of the read. In our standard run, we look for at least 5 contiguous bases (NUM_BASES) of the SMART adapter (SEQUENCE) at the 5' end of the read with no errors (MISMATCHES) , and hard clip those bases off the read.

Example:

TrimStartingSequence
INPUT=unaligned_tagged_filtered.bam
OUTPUT=unaligned_tagged_trimmed_smart.bam
OUTPUT_SUMMARY=adapter_trimming_report.txt
SEQUENCE=AAGCAGTGGTATCAACGCAGAGTGAATGGG
MISMATCHES=0
NUM_BASES=5

PolyATrimmer

This Drop-seq program is the second sequence cleanup program designed to trim away trailing polyA tails from reads. It searches for at least 6 (NUM_BASES) contiguous A's in the read with 0 mismatches (MISMATCHES), and hard clips the read to remove these bases and all bases 3' of the polyA run.

Example:

PolyATrimmer
INPUT=unaligned_tagged_trimmed_smart.bam
OUTPUT=unaligned_mc_tagged_polyA_filtered.bam
OUTPUT_SUMMARY=polyA_trimming_report.txt
MISMATCHES=0
NUM_BASES=6
USE_NEW_TRIMMER=true

SamToFastq

Now that your data has had the cell and molecular barcodes extracted, the reads have been cleaned of SMARTSeq primer and polyA tails, and the data is now unpaired reads, it's time to align. To do this, we extract the FASTQ files using Picard's SamToFastq program.

Example:

java -Xmx4g -jar /path/to/picard/picard.jar SamToFastq
INPUT=unaligned_mc_tagged_polyA_filtered.bam
FASTQ=unaligned_mc_tagged_polyA_filtered.fastq

Alignment - STAR

We use STAR as our RNA aligner. The manual for STAR can be found here. There are many potential aligners one could use at this stage, and it's possible to substitute in your lab's favorite. We haven't tested other aligners in methodical detail, but all should produce valid BAM files that can be plugged into the rest of the process detailed here.

If you're unsure how to create an indexed reference for STAR, please read the STAR manual.
Below is a minimal invocation of STAR. Since STAR contains a huge number of options to tailor alignment to a library and trade off sensitivity vs specificity, you can alter the default settings of the algorithm to your liking, but we find the defaults work reasonably well for Drop-seq. Be aware that STAR requires roughly 30 gigabytes of memory to align a single human sized genome, and 60 gigabytes for our human/mouse reference.

Example:

```
/path/to/STAR/STAR
--genomeDir /path/to/STAR_REFERENCE
--readFilesIn unaligned_mc_tagged_polyA_filtered.fastq
--outFileNamePrefix star
```

SortSam

This picard program is invoked after alignment, to guarantee that the output from alignment is sorted in queryname order. As a side bonus, the output file is a BAM (compressed) instead of SAM (uncompressed.)

Example:

```
java -Xmx4g -jar /path/to/picard/picard.jar SortSam
I=starAligned.out.sam
O=aligned.sorted.bam
SO=queryname
```

MergeBamAlignment

This Picard program merges the sorted alignment output from STAR (ALIGNED_BAM) with the unaligned BAM that had been previously tagged with molecular/cell barcodes (UNMAPPED_BAM). This recovers the BAM tags that were "lost" during alignment. The REFERENCE_SEQUENCE argument refers to the fasta metadata file.

We ignore secondary alignments, as we want only the best alignment from STAR (or another aligner), instead of assigning a single sequencing read to multiple locations on the genome.

Example:

```
java -Xmx4g -jar /path/to/picard/picard.jar MergeBamAlignment
REFERENCE_SEQUENCE=my_fasta.fasta
UNMAPPED_BAM=unaligned_mc_tagged_polyA_filtered.bam
```

ALIGNED_BAM=aligned.sorted.bam
OUTPUT=merged.bam
INCLUDE_SECONDARY_ALIGNMENTS=false
PAIRED_RUN=false

TagReadWithGeneExon
This is a Drop-seq program that adds a BAM tag "GE" onto reads when the read overlaps the exon of a gene. This tag contains the name of the gene, as reported in the annotations file. You can use either a GTF or a RefFlat annotation file with this program, depending on what annotation data source you find most useful. This is used later when we extract digital gene expression (DGE) from the BAM.

Example:
TagReadWithGeneExon
I=merged.bam
O=star_gene_exon_tagged.bam
ANNOTATIONS_FILE=${refFlat}
TAG=GE

Updates to TagReadWithGeneExon (V2)
We have updated and re-written how reads are tagged with functional annotations in V 2.0 of the dropseq toolkit. In V1, reads received two BAM tags when a read overlapped the exon of a gene. The GE tag specified the gene that overlapped the read, while GS specified which strand the gene was on. This information allows DigitalExpression and other programs to decide if they want to consider reads that are on the same strand as the gene, or run without regard to strand.

A typical read on that overlaps a gene might have the following tags, indicating the read overlapped an exon of GENE_A, and was on the positive strand:

H53FWBGXX150403:1:11307:13550:9549 0 1 29658 1 60M * 0 0 CTGCCTTCCCCTCAAGCTCAGGGCCAAGCTGTCCGCCAACCTCGGCTCCTCCGGGCAGCC 7FFFFFFFFFFFFFFFFFFF.FFFFFFFFFFFAFFFFFFFFFFA.FFFF<FFFFAAAAA XC:Z:TTGTCATGTCAC **GE:Z:GENE_A** XF:Z:CODING PG:Z:STAR.1 RG:Z:H53FW.1 H:i:4 NM:i:0 XM:Z:GCAAACCT UQ:i:0 AS:i:59 **GS:Z:+**

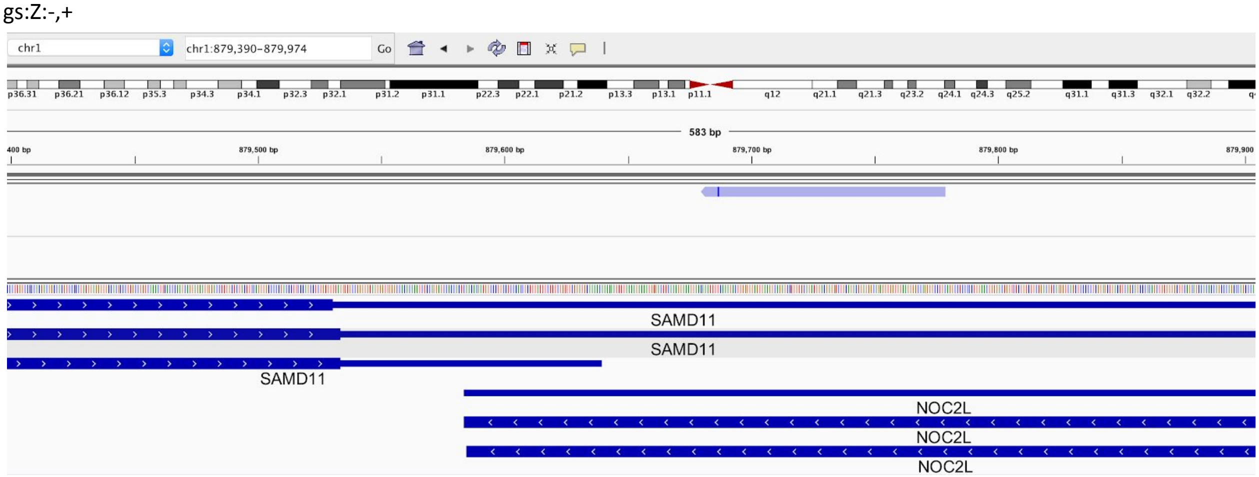
This functionality has been retained exactly as it was implemented in a newly distributed program TagReadWithGeneExonFunction. We've done this in case other users need to retain backwards compatibility with any analysis they may have implemented.

TagReadWithGeneFunction (replacement for TagReadWithGeneExon)
Our replacement for TagReadWithGeneExon is TagReadWithGeneFunction. This program provides a more flexible and informative set of tags for reads that allow downstream programs to measure not only digital expression of reads that overlap exons, but can leverage reads that introns as well. This program provides 3 tags for each read, **gn** [gene name], **gs** [gene strand] and **gf** [gene function]. These tags can

have more than one value, and the values are comma separated. These tags can also co-exist with the original tagger (TagReadWithGeneExon) as the tag names are different, so if you use those tags for other purposes, you can tag your BAM with both taggers.

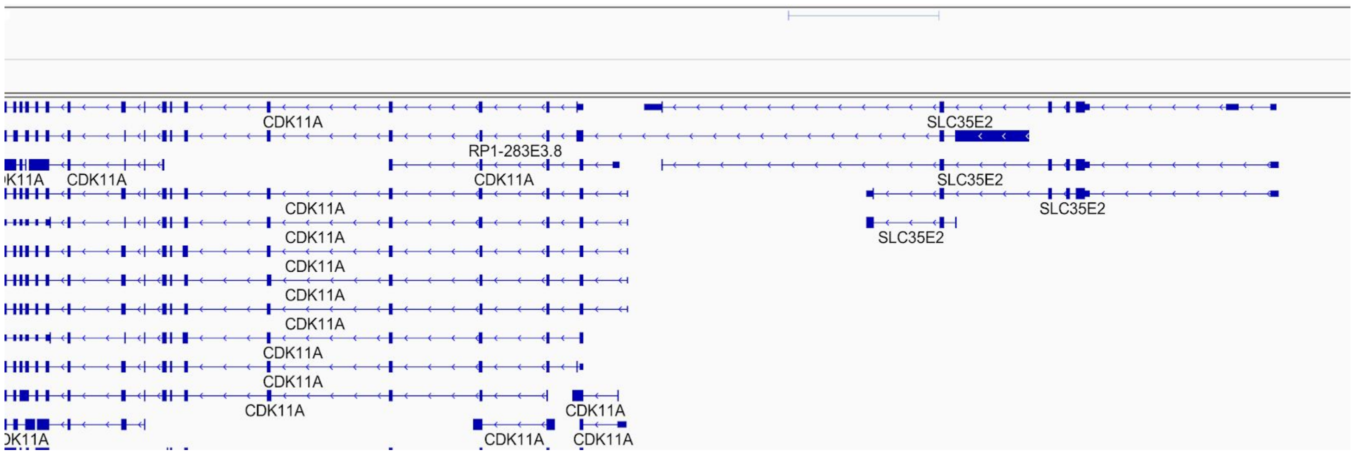
Example Invocation (The call to TagReadWithGeneFunction is the same as TagReadWithGeneExon)
TagReadWithGeneFunction
I=merged.bam
O=star_gene_exon_tagged.bam
ANNOTATIONS_FILE=${refFlat}

Below is an example read using the new tagger:
HFWN3DMXX:1:2133:1949:24283 16 1 879682 255 98M * 0 0
AATTTCCAAAGACTTGGGGGAGTGAAGGCAGAGCCTGGTGCAGATGGACGAGGTCTGCAGACGGAGGGCAGAGGTGGTGGAAGGGGCCA
GGGGCCTGC FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
XC:Z:CTACGTCCACATGACT MD:Z:5T92 XF:Z:CODING PG:Z:STAR.3W RG:Z:HFWN3.1
XG:Z:SAMD11,NOC2L NH:i:1 NM:i:1 XM:Z:GAAGGATAAA UQ:i:37 AS:i:94 gf:Z:CODING,UTR gn:Z:NOC2L,SAMD11
gs:Z:-,+



The gs, gn, and gf tags all have the same number of values. They are interpreted as a trio of values that describe the gene the read overlaps, the strand the gene is on, and the functional annotation of the gene at that position. In the example above, the read overlaps both **NOC2L** on the negative strand and completely overlaps an exon. The read also overlaps **SAMD11** on the positive strand. The read is on the negative strand (from the bitflag on the read of 16), so standard DGE would interpret this as expression of **NOC2L.**

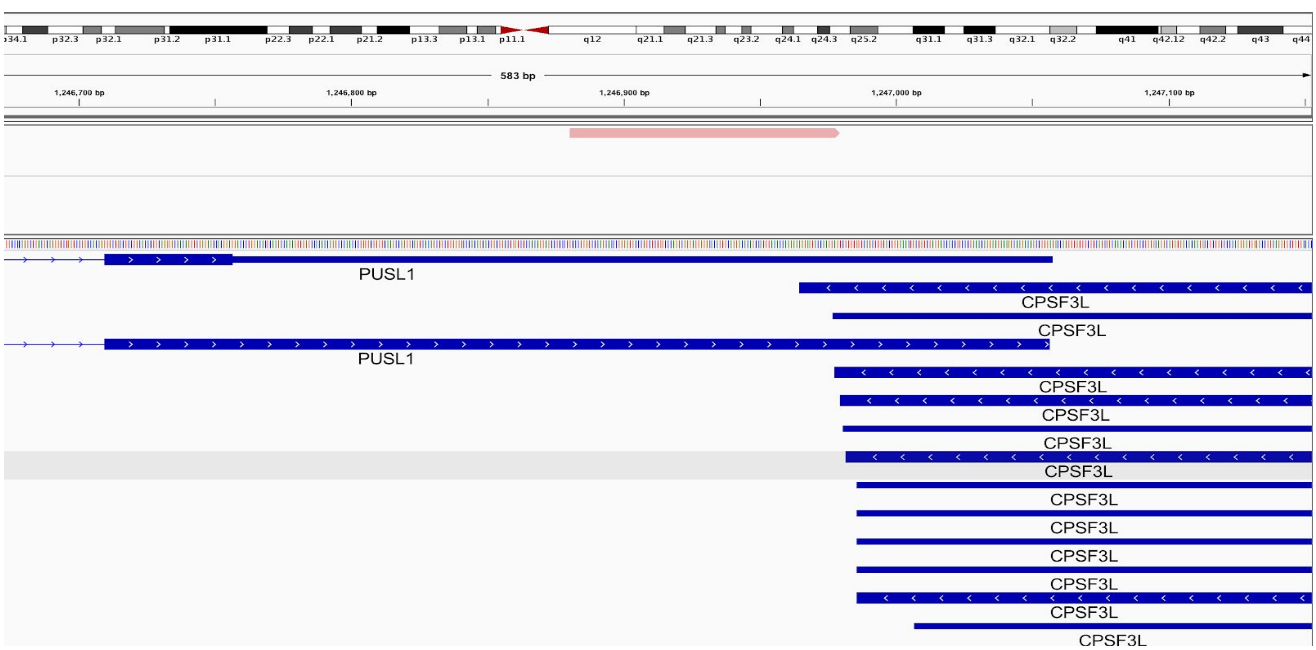
Example 2:
HFWNNDMXX:2:2220:15085:21292 16 1 1661100 255 2S20M5009N76M * 0 0
CCGAGCCACCGCAGCCGGTCTTCTGAAAGTCACCGGGGAGATTTTCCCCATGAGGGCGTACGCCGTGACGCTCTGAAGGTGGAACAGGACT
CCGTCTG FFFFFFFFFFFFFFFFFFF,FFFFFF:FFFFFFFFFFFFFFFFFFFFF:FFFFFFFFFFFFFFFFFFFFFFFFF:FFFFFF:FFF:FFFFFFFFFFFF
XC:Z:TCAGGATCAGCAGTTT MD:Z:4T91 XF:Z:CODING
PG:Z:STAR.3O RG:Z:HFWNN.2.B XG:Z:RP1-283E3.8,SLC35E2 NH:i:1 NM:i:1 XM:Z:ACATGCCGCG UQ:i:37
AS:i:91 gf:Z:CODING,INTRONIC,CODING,INTRONIC gn:Z:RP1-283E3.8,RP1-283E3.8,SLC35E2,SLC35E2 gs:Z:-,-,-,-



This read is a bit more "interesting", due to the overlapping gene annotations. Both SLC35E2 and RP1-283E.8 appear to share the same exon, though in different splicing contexts. The read is mapped as a split read, where part of the read is gapped and splices to a different location, which is common when mapping exon-exon junctions. The other part of the read appears to splice in the middle of the intron for both genes. DGE will interpret this read as ambiguous, as it can be assigned to either gene.

Example 3:

HFWNNDMXX:2:2114:15917:25019 0 1 1246881 255 98M * 0 0
GCCCGGGTCCCAGCACCCTGGATGCCCGTCTCTGTCCCAGGCGGGATGGGGCACAGTGCAGGACACAGCCATGTACACCAAGAAGAGAGTA
CCAAGTA F:FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
XC:Z:TGCCAAAGTCGCTTTC MD:Z:98 XF:Z:CODING PG:Z:STAR.15
RG:Z:HFWNN.2.A XG:Z:CPSF3L,PUSL1 NH:i:1 NM:i:0 XM:Z:GTATGATTGA UQ:i:0 AS:i:96
gf:Z:CODING,INTERGENIC,CODING gn:Z:CPSF3L,CPSF3L,PUSL1 gs:Z:-,-,+

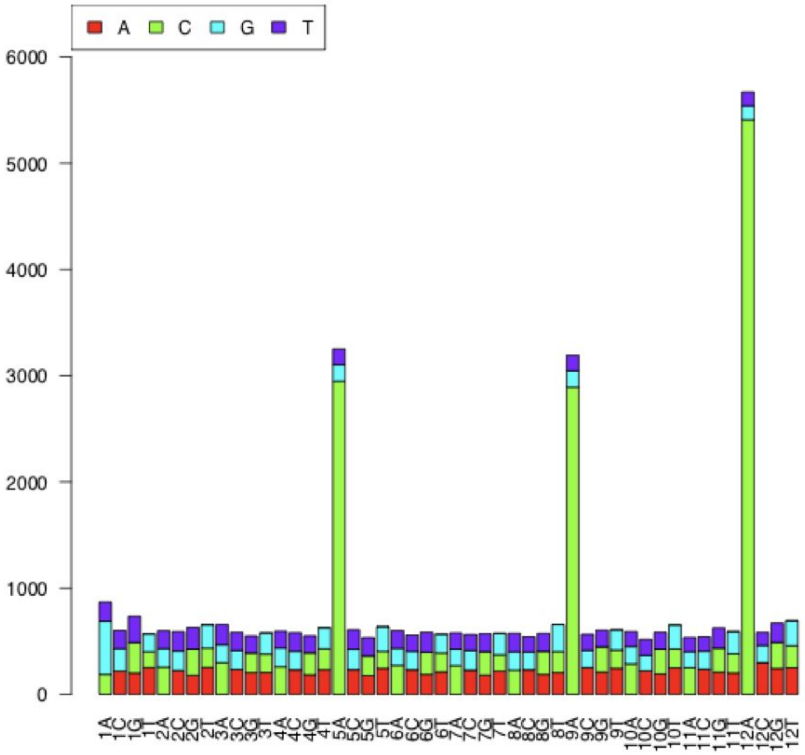


This read maps to two genes, CPSF3L and PUSL1. The read is on the positive strand, as is PUSL1, so the read would be assigned to that gene. If you were extracting expression on the opposite strand from the gene, then the read would be assigned both coding and intergenic (outside the bounds of the gene) portions of CPSF3L. Since the read wasn't assigned to only CODING+UTR portions of the transcript, under the standard functional types fort DGE, the read would be ignored. Reads must be entirely contained within the requested functional types to be counted. If you wanted to go wild, (and who doesn't?), you could request DGE to extract the antisense transcript and include intergenic regions by using STRAND_STRATEGY=ANTISENSE LOCUS_FUNCTION_LIST=INTERGENIC. That would generate a UMI for this cell barcode on the CPSF3L gene. Hopefully this demonstrates the flexibility of our new approach to tagging reads.

DetectBeadSubstitutionErrors - Detecting and repairing substitution errors in cell barcodes

In previous chemgenes bead lots, we have observed non-random patterns of substitution changes at hamming edit distance=1 between pairs of barcodes that appear to be related. Given many of these cell barcodes may in fact be multiple cell barcodes that reside on a single physical bead, it makes sense to combine the reads across these barcodes. However, barcodes can appear to be related to each other at hamming distance=1 by chance due to sequencing/PCR errors, or because we sample a significant subset of the total available cell barcode space and happen to observe two barcodes that are truly independent, but very similar in sequence. The challenge is to combine related cell barcodes together that have arisen from the same bead, while avoiding capricious collapse of other pairs of cell barcodes that are related by chance.

Errors that occur at the synthesis level ought to be systemic - the same barcode position and base substitution pattern from an intended sequence to a related sequence (for example: position 5 of many cell barcodes change from A to C) should be consistent across an entire experiment at some substitution rate. The higher the rate of substitution, the more frequently the related sequence will be observed. To determine where these events take place, we survey the set of cell barcodes with at least 20 transcripts, and exhaustively look for pairs of barcodes that are related at hamming distance=1. For each pair of barcodes, we assume the more frequently observed barcode to be the "intended" sequence. We filter pairs of barcodes so that smaller barcodes are unambiguously related to one and only one intended sequence. By building up this set of barcodes, we can observe patterns in the substitution events that are biased to certain bases and positions in the synthesis reaction.



The X axis describes the position of the substitution and the intended base. The color of the stacked column indicates the base that is substituted at that position.

There are clear patterns of substitution events that occur in the data above at some positions, as well as a lower level of stochastic changes that occur at every position. By looking for a dominant base change (>80% of events) at each position and intended sequence, a subset of all possible base substitution events can be selected. In this case, there are A->C substitution events at bases 1,2,3,4,5,9,12. We select barcodes that contain these specific substitution bases and positions to perform repair at. We remove the smaller neighbors that are related to multiple intended sequences, or do not fit the substitution pattern observed above. These patterns are discovered on an experiment by experiment basis as part of the cleanup process.

These results are repeatable across many experiments using the same bead lot, and differ across bead lots.

Example:
DetectBeadSubstitutionErrors
I=my.bam
O=my_clean_subtitution.bam
OUTPUT_REPORT=my_clean.substitution_report.txt

DetectBeadSynthesisErrors - Detecting and repairing barcode indel synthesis errors

In June 2015, we noticed that a recently purchased batch of ChemGenes beads generated a population of cell barcodes (about 10-20%) with sequences that shared the first 11 bases, but differed at the last base. These same cell barcodes also had a very high percentage of the base "T" at the last position of the UMI. Based on these observations, we concluded that a percentage of beads in the lot had not undergone all twelve split-and-pool bases (perhaps they had stuck to some piece of equipment or container, and the been re-introduced after the missing synthesis cycle). Thus, the 20-bp Read 1 contained a mixed base at base 12 (in actuality, the first base of the UMI) and a fixed T-base at base 20 (in actuality, the first base of the polyT segment).

To correct for this, we generated DetectBeadSynthesisErrors, which identifies cell barcodes with aberrant "fixed" UMI bases. If only the last UMI base is fixed as a T, the cell barcode is corrected (the last base is trimmed off) and all cell barcodes with identical sequence at the first 11 bases are merged together. If any other UMI base is fixed, the reads with that cell barcode are discarded.

UPDATE

More recently (Fall 2017), we observed that it was possible to not only discover cell barcodes where the UMIs were biased to T at the last base, but in many cases to discover what the original barcode sequence was. During synthesis, these incorporation errors often occur incompletely - a base is missing at a certain position, and elsewhere in the experiment, it's possible to recover the original "intended" sequence as another cell barcode where the UMIs do not experience the T bias, and the 2 cell barcodes are related by a 1 base pair insertion/deletion event. The intended sequence has 4 "neighbor" barcodes at an indel distance of 1 (and if they have few UMIs, not all 4 neighbors will be detected), and those neighbors are all related to each other by a substitution edit distance of 1 at the last base of their sequence.. Each of these neighbors also has high T bias at the last base of their UMIs. By looking at the number of UMIs observed by the intended and neighbor sequences, it's possible to calculate the rate at which the base was not incorporated into the sequence. For example, if the intended sequence is quite small, and the neighbors are relatively large, then the base was not incorporated at a high rate.

This allows us to validate which UMI biased cell barcodes should properly be merged into an intended sequence, which is a more stringent repair process than what we'd previously employed. Because of this, we've removed the requirement to estimate the number of cells in the experiment. We repair all cell barcodes with at least 20 UMIs.

Below is an example of a few intended sequence / neighbor barcodes and their relationships. When this software is run, these parameters (and many others) are emitted as a result. The neighbor sequences are colon separated.

Example	Intended Sequence	Neighbor Sequences	Deleted base	Deleted base position	Non incorporation rate
1	TGATGCACGAGG	TGATCACGAGGA:TGATCACGAGGC:TGATCACGAGGG:TGATCACGAGGT	G	5	0.01
2	CTCCGAACTGCC	CTCCGAACGCCA:CTCCGAACGCCC:CTCCGAACGCCG:CTCCGAACGCCT	T	9	0.95
3	CCCTCGTTAGAT	CCTCGTTAGATA:CCTCGTTAGATC:CCTCGTTAGATT	C	3	0
4	<NA>	CCCGCAGCTTGA:CCCGCAGCTTGC:CCCGCAGCTTGG:CCCGCAGCTTGT	<NA>	<NA>	<NA>

1. An intended sequence where all 4 neighbors are discovered. Non-incorporation rate is low, so the intended cell barcode has many UMIs relative to the neighbor cell barcodes.
2. An intended sequence where all 4 neighbors are discovered. Non-incorporation rate is high, so the intended cell barcode has few UMIS relative to the neighbor cell barcodes.
3. Only 3 neighbors are discovered. Fewer than 4 neighbors being discovered can occur when the neighbors have ~ 25 UMIs (the smallest cell barcode we look at is 20), and one of the neighbors has fewer UMIs so is not reported. The missing sequence is CCTCGTTAGATG, as A/C/T are found.
4. 4 related neighbors are found, but the intended sequence is not discovered. This occurs when the base non-incorporation rate is very high , and occurs in almost every copy of the barcode on the bead. Because the intended sequence is not discovered, the other properties can not be determined.

Example:

DetectBeadSynthesisErrors
I=my_clean_subtitution.bam
O=my_clean.bam
REPORT=my_clean.indel_report.txt
OUTPUT_STATS=my.synthesis_stats.txt
SUMMARY=my.synthesis_stats.summary.txt
PRIMER_SEQUENCE=AAGCAGTGGTATCAACGCAGAGTAC

This program reads in the BAM file, and looks at the distribution of bases at each position of all UMIs for a cell barcode. It detects unusual distributions of base frequency, where a base with >=80% frequency at any position is detected as an error. Barcodes with less than 20 total UMIs are ignored. There are a number of different errors that are categorized:

1. SYNTHESIS_MISSING_BASE - 1 or more bases missing from cell barcode, resulting in fixed T's at the end of UMIs. This counts the maximum number of fixed sequential T's in the UMIs at the end. This error type is cleaned up by the software for situations where there is a single base

missing, and is by far the most common error. The fix involves inserting an "N" base before the last cell barcode base, effectively shifting the reading frame back to where it should be. This will both collapse these beads back together in further analysis, as well as repair the UMIs for these bead barcodes. **[Note as of V2, we no longer insert an N into sequences to repair them if we're able to determine the intended sequence. In those cases we use the intended sequence instead.]**

2. SINGLE_UMI_ERROR - At each position of the UMIs, the base distribution is highly skewed, i.e. at each position, a single base appears in >= 80% of the UMIs for that cell. There's no fix for this currently. Cell barcodes with this property are dropped. These cells have the interesting property that the number of genes and transcripts are at a close to 1:1 ratio, as there's generally only 1 UMI for every gene.

3. PRIMER_MATCH - Same as SINGLE_UMI_ERROR, but in addition the UMI perfectly matches one of the PCR primers. These cell barcodes are dropped. These errors are only detected if a PRIMER_SEQUENCE argument is supplied.

4. 4) OTHER - UMIs are extremely skewed towards at least one base (and not T at the last base), but not at all 8 positions. These cell barcodes are dropped.

The file my.synthesis_stats.txt contains a bunch of useful information:

1. CELL_BARCODE - the 12 base cell barcode
2. NUM_UMI - the number of total umis observed
3. FIRST_BIASED_BASE - the first base position where any bias is observed. -1 for no detected bias
4. SYNTH_MISSING_BASE - as #3 but specific to runs of T's at the end of the UMI
5. ERROR_TYPE - see error type definitions above
6. For bases 1-8 of the UMI, the observed base counts across all UMIs. This is a "|" delimited field, with counts of the A,C,G,T,N bases.

The file my.synthesis_stats.summary.txt contains a histogram of the SYNTHESIS_MISSING_BASE errors, as well as the counts of all other errors, the number of total barcodes evaluated, and the number of barcodes ignored.

End of Alignment

At this point, the alignment is completed, and your raw reads have been changed from paired reads to single end reads with the cell and molecular barcodes extracted, cleaned up, aligned, and prepared for DGE extraction.

Going with the flow - using Unix pipes to simplify alignment

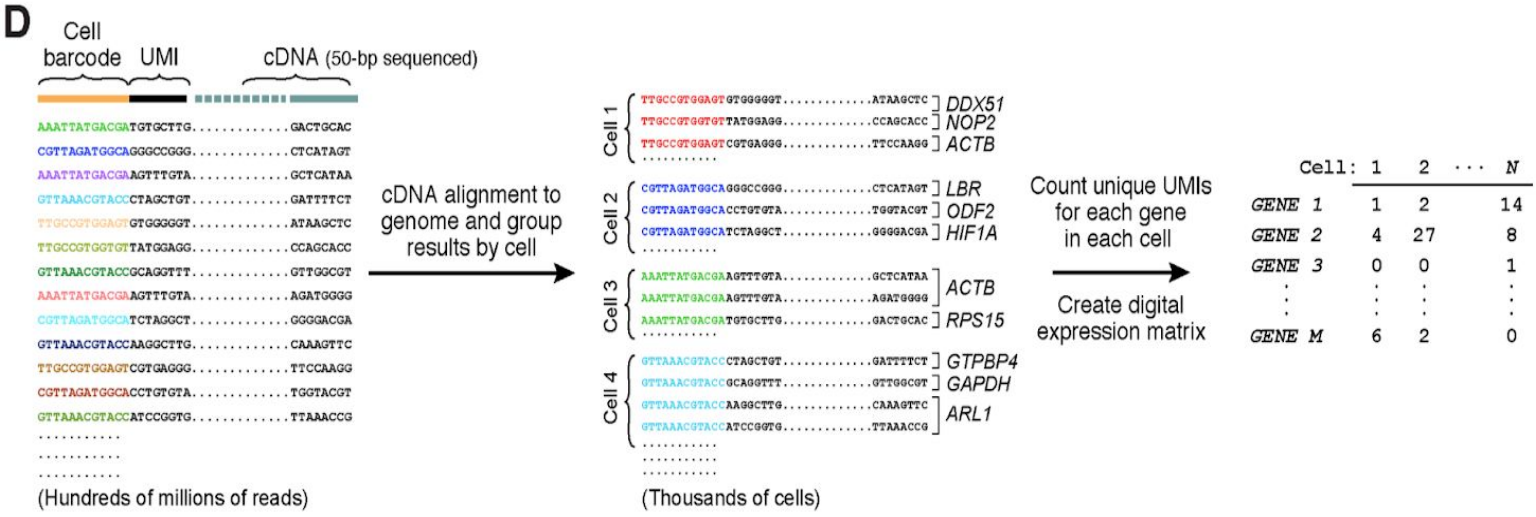
If you're on a Unix or OS X operating system, you may be familiar with pipes. Drop-seq programs extend the Picard API, and so like Picard are able to use pipes to redirect output from one program to the next. Why is this useful? It's a little bit faster, but more importantly it saves a significant amount of disk space by not generating a large number of temporary files, as the examples above have. It also simplifies writing pipelines, as there are fewer named files - intermediate data flows through the pipeline without being saved. The tradeoff is that executing several programs in a pipeline requires more RAM and more processing power, so if your computer does not have a lot of RAM and lots of processors, this might not be useful.

There are some limitations to the amount of pipelining that can be done, because some files must be read more than once, and because STAR does not have the ability to write to standard output. The following steps may be pipelined:

- Pre-alignment tagging and trimming. The final result of these steps must be saved to be input to both SamToFastq and MergeBamAlignment.
    - TagBamWithReadSequenceExtended (one or more times)
    - FilterBam
    - TrimStartingSequence
    - PolyATrimmer
- Alignment (STAR does not support output to a pipe):
    - SamToFastq
    - STAR
- Merging and tagging aligned reads:
    - MergeBamAlignment
    - TagReadWithGeneFunction

The bead-repair programs make multiple passes over the input so they can't be pipelined.

Overview of DGE extraction



To digitally count gene transcripts, a list of UMIs in each gene, within each cell, is assembled, and UMIs within edit distance = 1 are merged together. The total number of unique UMI sequences is counted, and this number is reported as the number of transcripts of that gene for a given cell.

Digital Gene Expression

Extracting Digital Gene Expression (DGE) data from an aligned library is done using the Drop-seq program DigitalExpression. The input to this program is the aligned BAM from the alignment workflow. There are two outputs available: the primary is the DGE matrix, with each a row for each gene, and a column for each cell. The secondary analysis is a summary of the DGE matrix on a per-cell level, indicating the number of genes and transcripts observed.

Primary Output Example:

GENE	ATCAGGGACAGA	AGGGAAAATTGA	TTGCCTTACGCG	TGGCGAAGAGAT	TACAATTAAGGC
LOXL4	0	0	0	0	0
PYROXD2	1	0	1	1	0
HPS1	23	12	9	8	3
CNNM1	0	2	1	0	0
GOT1	22	6	7	9	3

Summary Output Example:

CELL_BARCODE	NUM_GENES	NUM_TRANSCRIPTS
ATCAGGGACAGA	12128	232831
AGGGAAAATTGA	12161	185418
TTGCCTTACGCG	10761	173547
TGGCGAAGAGAT	10036	108545
TACAATTAAGGC	9889	99771
CTAAGTAGCTTT	9244	91563

Long output Example (new for V2):

CELL	GENE	UMI_COUNT
ATCAGGGACAGA	HPS1	23
ATCAGGGACAGA	GOT1	22
ATCAGGGACAGA	PYROXD2	1
AGGGAAAATTGA	HPS1	12
AGGGAAAATTGA	GOT1	6

This file is ordered by the list of cell barcodes (input cell barcode file or based on number of reads per cell), then the number of UMIs per gene, then alphabetically by gene when they have the same number of UMIs. There are no entries when a cell does not have expression of a gene.

DGE Extraction Options:

There are a large number of options in the DGE program, as we've performed large amounts of experimentation with the outputs to this program. Most of these parameters have default settings, and

are the correct setting for a standard Drop-seq experiment. Outlined below are some of the parameters that you might change.

READ_MQ The minimum map quality of a read to be used in the DGE calculation. For aligners like STAR, the default (10) is higher than what's needed to eliminate all multi-mapping reads. If you use a different aligner, you might want to set a different threshold.

EDIT_DISTANCE. By default we collapse UMI barcodes with a hamming distance of 1.

RARE_UMI_FILTER_THRESHOLD This is an implementation of the rare UMI filter implemented by Islam, et al. We leave this off by default, and use edit distance collapse instead. If desired, one can set EDIT_DISTANCE=0 and enable this filter instead at some threshold, like 0.01.

Options for selecting sets of cells

When running DGE, we don't select every cell barcode observed. This is because the aligned BAM can contain hundreds of thousands of cell barcodes; most reads will be on either STAMPs (beads exposed to a cell in droplets) or "empties" (beads that were exposed only to ambient RNA in droplets). There will also be a lot of cell barcodes with just a handful of reads. Because a huge matrix might be difficult to work with, these options limit the number of cell barcodes that are emitted by DGE extraction. *You must use one of these options.*

MIN_NUM_GENES_PER_CELL. DigitalExpression runs a single iteration across all data, and selects cells that have at least this many genes.

MIN_NUM_TRANSCRIPTS_PER_CELL. DigitalExpression runs a single iteration across all data, and selects cells that have at least this many transcripts. (Finally bugfixed and working in V2.0.0!)

NUM_CORE_BARCODES. DigitalExpression counts the number of reads per cell barcode (thresholded by READ_MQ), and only includes cells that have at least this number of reads.

CELL_BC_FILE. Instead of iterating over the BAM and discovering what cell barcodes should be used, override this with a specific subset of cell barcodes in a text file. This file has no header and a single column, containing one cell barcode per line. Since this option doesn't have to iterate through the BAM to select barcodes, DGE extraction is significantly faster when using this option.

Functional annotations and strand selection, NEW for V2.0.0:

Along with the changes to how reads are tagged with functional annotations, DGE and similar programs are now able to extract these enhanced sets of tags. There are two main parameters to use:

STRAND_STRATEGY:

The strand strategy decides which reads will be used by analysis based on the strand of the read and the strand of the gene. The SENSE strategy requires the read and annotation to be on the same strand. The

ANTISENSE strategy requires the read and annotation to be on opposite strands. The BOTH strategy is permissive, and allows the read to be on either strand.

LOCUS_FUNCTION_LIST:
This is a list of functional annotations that should be used to include reads in analysis. The default is include reads in DGE analysis where the read entirely overlaps the CODING and UTR portions of a gene. This is slightly more conservative than DGE V1, which allowed reads that only partially overlap an exon to be counted. Changing the list of annotations allows for different sorts of expression data to be extracted.

Example:
In this example, we extract the DGE for the top 100 most commonly occurring cell barcodes in the aligned BAM, using CODING+UTR regions on the SENSE strand.

DigitalExpression
I=out_gene_exon_tagged.bam
O=out_gene_exon_tagged.dge.txt.gz
SUMMARY=out_gene_exon_tagged.dge.summary.txt
NUM_CORE_BARCODES=100

Example INTRONIC+CODING:
If you want to simply add additional annotations to CODING+UTR, specifying LOCUS_FUNCTION_LIST adds to the list. For example, we add intronic expression, and coding is already specified as the default.

DigitalExpression
I=out_gene_exon_tagged.bam
O=out_gene_exon_tagged.dge.txt.gz
SUMMARY=out_gene_exon_tagged.dge.summary.txt
NUM_CORE_BARCODES=100
LOCUS_FUNCTION_LIST=INTRONIC.

Example INTRONIC ONLY:
There's a bit of a "gotcha" in how this is specified by to the program (this comes from the Picard's API for command line argument interpretation.) If you want to specify INTRONIC only expression, you first need to clear the list of functional annotations by giving a value of null, then add your additional values:

DigitalExpression
I=out_gene_exon_tagged.bam
O=out_gene_exon_tagged.dge.txt.gz
SUMMARY=out_gene_exon_tagged.dge.summary.txt
NUM_CORE_BARCODES=100
LOCUS_FUNCTION_LIST=null LOCUS_FUNCTION_LIST=INTRONIC.

Cell Selection

A key question to answer for your data set is how many cells you want to extract from your BAM. One way to estimate this is to extract the number of reads per cell, then plot the cumulative distribution of reads and select the "knee" of the distribution.

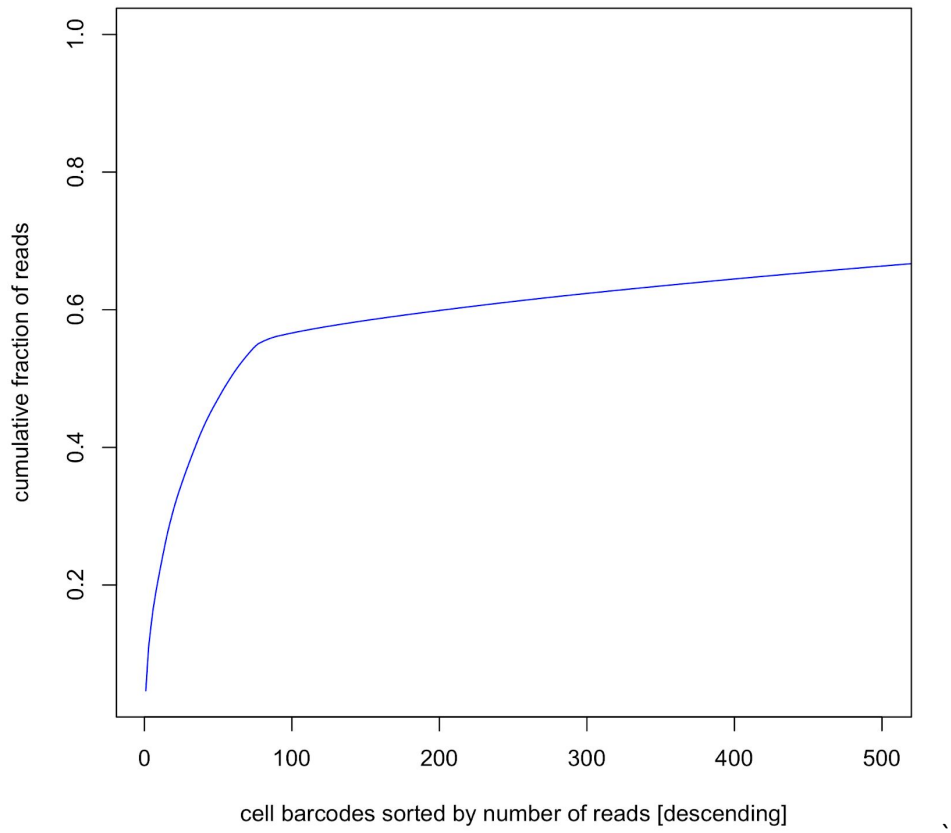
We provide a tool to extract the reads per cell barcode in the Drop-seq software called BAMTagHistogram. This extracts the number of reads for any BAM tag in a BAM file, and is a general purpose tool you can use for a number of purposes. For this purpose, we extract the cell tag "XC":

Example:

```
BAMTagHistogram
I=out_gene_exon_tagged.bam
O=out_cell_readcounts.txt.gz
TAG=XC
```

Once we run this program, a little bit of R code can create a cumulative distribution plot. Here's an example using the 100 cells data from the Drop-seq initial publication (Figures 3C and 3D):

```
a=read.table("100cells_numReads_perCell_XC_mq_10.txt.gz", header=F, stringsAsFactors=F)
x=cumsum(a$V1)
x=x/max(x)
plot(1:length(x), x, type='l', col="blue", xlab="cell barcodes sorted by number of reads [descending]",
ylab="cumulative fraction of reads", xlim=c(1,500))
```



In this example, the number of STAMPs are the number of cell barcodes to the left of the inflection point; to the right of the inflection point are the empty beads that have only been exposed to ambient RNA. Figure S3A of Macosko et al., 2015 provides additional justification and explanation for how we identify the number of cells sequenced.

Mixed-species plots

To create the mixed species plots used in the paper, we suggest the following steps:

1. Align your data to a mixed species reference. There is metadata at the bottom of one of the pages of our GEO submission
2. Determine how many cells are in your BAM. See **Cell Selection** in this document and the BAMTagHistogram program. Put that list of cell barcodes in a file that has a single column of cell barcodes, 1 per line.
3. At this point, if you are using our human/mouse metadata, your BAM has chromosomes that are prepended with HUMAN or MOUSE, i.e.: HUMAN_11. Filter your BAM into 2 organism specific BAMs using FilterBam with the argument REF_SOFT_MATCHED_RETAINED=HUMAN or REF_SOFT_MATCHED_RETAINED=MOUSE (this is about as fancy as running grep on your BAM.)

4. Run DigitalExpression on each organism specific BAM with the CELL_BC_FILE argument, using the file generated in step #2.
5. You now have two summary files that have the number of genes/transcripts contained by each cell, in an organism specific manner. Merge them into one file and plot.

Conclusion

With successful execution of our software you have hopefully transformed a pile of hundreds of millions of sequence reads into a digital expression matrix that has genome-wide expression measurements (digital counts) for each gene in each individual cell.

What to do next? We expect analysis of massive single-cell expression data to become a lively field. We think very highly of the Seurat package developed by our colleague Rahul Satija. We used Seurat to perform all of the downstream analyses (cell clustering, etc) in the Cell paper. Seurat is available on Rahul's web site (http://www.satijalab.org/seurat.html), where Rahul will also have protocols for the specific analyses in the paper

But what if everything doesn't go perfectly?

One of the big challenges with releasing a new software toolkit to the world is that people will always do things you didn't anticipate, with data sets you never imagined. While we feel the Drop-seq software produces the computationally correct (at least to our intentions) answers, it's possible that you will discover a bug, or documentation of a particular software parameter will be unclear.

If you find part of this document unclear, let us know and we'll do our best to update it and add clarity. If parameters of our software have unclear documentation, let us know which ones are unclear, and we'll do our best to buff up those descriptions.

If you run into software behavior you think is a bug, then you can help to be part of the solution. To do this, you'll need to give us the following information

- The program you were running, and the exact command line arguments you supplied to that program
- The console output of the program invocation
- A small test data set that can replicate the problem you observed
- The behavior that you think was faulty, and if possible what you expected to see. This can be very useful when a computation produces an answer that doesn't make sense.
- We have a public githib repository at https://github.com/broadinstitute/Drop-seq. If you're a programmer, you can submit pull requests to us to fix bugs or add additional capabilities.