week02: read mapping with kallisto

goals this week

- an introduction to **kallisto**, an innovative, important, and still relatively new method for RNA-seq read mapping and transcript quantitation;
- our first time installing and running a command line application for biological data analysis, and controlling it with Python and Jupyter notebook;
- our first time dealing with a couple of important biological sequence data formats: FASTA for sequences, FASTQ for reads and their base-calling "quality values".
- a peek ahead at some stuff we'll get deeper into in weeks ahead, as we see how *kallisto* combines its read mapping algorithm with using a likelihood model and expectation maximization for quantitation.

read mapping: historical overview

In 2006, Illumina Solexa sequencers were introduced, capable of cheaply obtaining a large number of very short reads (20-35nt). Up until then, DNA sequencing had largely been used to assemble sequences of genes and genomes. Using short reads for genome assembly is problematic, because assembly requires overlap. But many people quickly realized that with an abundance of cheap, short reads, you could use sequencing as a counting assay; you could quantitate the abundance of individual sequences

in any DNA/RNA population by sequencing unique tags from them.

There was an explosion in something-Seq technologies, marrying some kind of DNA/RNA enrichment method (often an existing one) to a high through sequencing readout. For example, ChIP-Seq (chromatin immunoprecipitation + sequencing) means using antibodies to purify protein/DNA complexes, to identify what DNA sequences are bound by a specific protein.

For studying RNA populations, various sequencing strategies had already been applied to discoverying new RNAs and quantitating their abundance, either by completely sequencing cloned cDNA libraries, or various ways of using strategies to generate fragments or tags, such as *expressed sequence tag* (EST) sequencing. When Illumina short read technology was inevitably applied to high throughput mRNA quantitation, it was called RNA-seq.

To convert an RNA-seq experiment of $>$ 40M short 25nt reads to a useful quantitation of each mRNA transcript in a cell, the first computational step is to "map" reads to the possible transcripts they came from. From mapped read counts, we can infer transcript abundances.

To map a 25nt read to a transcriptome or a genome, at first people used similarity search programs like BLAST to find significant (high-scoring) alignments. But BLAST, and programs like it, is designed to find distantly related sequences, differing by many millions of years of evolution. Its algorithms are tuned to the homology search problem. It requires a fair amount of compute power, and for mapping 40M reads at a time, the compute requirements were prohibitive.

But the 25nt read isn't just similar; it is expected to map *exactly* or at least near-exactly to its source, differing essentially only by basecalling errors made by the sequencing machine. This

energized a new front in computational sequence analysis: the development of **read mappers**, using fast exact-match text matching algorithms from computer science, as opposed to the more statistical inference-based algorithms used for homology search. Exact-match text searching algorithms based on **hashing**, **suffix trees** and **suffix arrays** became important, but these methods (though powerful and important) often came with large memory requirements.

the Burrows-Wheeler transform

An important milestone in 2009 was the adoption of algorithms based on a surprising, beautiful, and counterintuitive algorithm called the Burrows-Wheeler Transform (BWT). The BWT was originally conceived as a data compression algorithm, but turned out to also be useful as the basis for memory-efficient suffix-array-like text searches. Two influential BWT-based read mappers were introduced in 2009: BWA, from Heng Li and Richard Durbin, and Bowtie, from Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg.

spliced read mappers, transcriptome assembly

If you know what mRNAs (or mRNA isoforms) are being made from a genome, you can map reads to a **transcriptome** (the set of all RNAs). If you don't -- if you want to use RNA-seq to identify new RNAs, or alternative isoforms -- then you can map reads to a **genome**. But then you have to deal with the fact that reads that span a spliced intron will map discontinuously to the genome. To use read mapping to identify unexpected intron splicing events, your read mapper has to map both sides of the read separately, allowing an intron between them. An important spliced read mapper was introduced in 2009: TopHat, by Trapnell, Pachter, and Salzberg.

As sequencing read lengths increased, it became possible to contemplate **de novo transcriptome assembly**, either by mapping reads to a genome first, or by trying to directly assemble RNA-seq reads. Mapping-first approaches include Scripture and Cufflinks assembly-first approaches include ABySS and Trinity.

I'm just letting you know that these other important angles exist. *kallisto*, which is where we're headed, assumes a transcriptome is known.

quantitation with mapping uncertainty

Reads don't uniquely map to their source, for a variety of reasons. There are duplications and repetitive sequences (Alus and such), and alternative RNA processing creates overlapping mRNA isoforms that share exonic sequences. Short reads and base calling errors exacerbate read mapping uncertainty.

As we've seen, an important milestone in dealing with read mapping uncertainty came from Bo Li, Colin Dewey, and coworkers in 2010. Li and Dewey's approach introduced a generative statistical model, inference with expectation maximization, and TPM units for measuring transcript abundance. We've seen TPM units already, and we'll be seeing generative statistical models and expectation maximization in weeks to come.

k-mer hashing

As short reads got longer (150nt and 300nt reads are now common on Illumina platforms), the cost of aligning an entire read to a potential source was still a problem (an hour or so to map a typical RNA-seq sample). In many areas, including both read mapping, sequence-based population diversity studies, and sequence assembly, interest

grew in using algorithms based on fixed-length **k-mers**, short subsequences of length k; simple and efficient **hashing** algorithms for identifying matching k-mers; and so-called **de Bruijn graphs** for representing how overlapping k-mers stitch together into larger "contigs" of transcripts and genomes. (I'll define these terms below; *kallisto* uses all of these concepts.)

An important milestone was the 2014 introduction of Sailfish, by Rob Patro, Steve Mount, and Carl Kingsford. The Sailfish paper, building on previous work on fast k-mer hashing in a program called Jellyfish, emphasized the fact that you don't need to map an entire 50-150nt read to know where it came from. Mapping a k-mer, if the k-mer is long enough to be unique, is usually sufficient. You can just think in terms of *assignment* of a read to potential sources, rather than *alignment*. Sailfish showed that this "alignment-free" approach could accelerate RNA-seq mapping and quantitation by more than an order of magnitude, and inspired the development of *kallisto*.

the kallisto pseudoalignment algorithm

kallisto was published in 2016 by Nicolas Bray, Harold Pimentel, Pall Melsted, and Lior Pachter, after being introduced a while before in a paper in the bioarxiv. The key idea is **pseudoalignment**. Like Sailfish, kallisto uses fast k-mer hashing to identify matching k-mers in a read and a transcriptome. kallisto uses a set of k-mer matches for each read to deduce the set of possible transcripts that the read could come from: it calls this assignment a *pseudoalignment*. (kallisto assumes that the transcriptome is known: you provide an input file with the sequences of all the possible RNA isoforms that you think are in play. We'll call this set of transcript sequences T.)

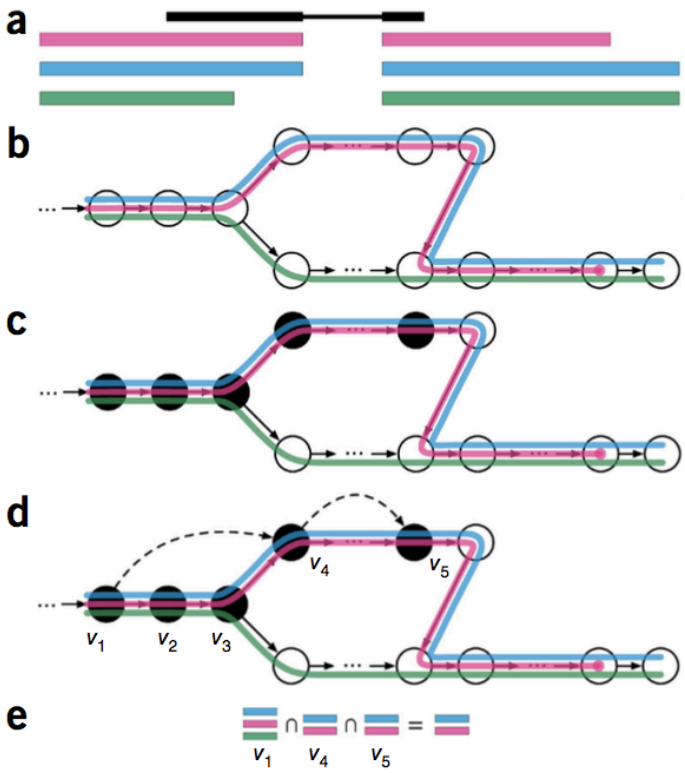


Figure 1 from the kallisto paper is shown above. (a) shows a read (black), and three mRNA isoforms (colors). (b) shows a toy *colored de Bruijn graph* for the transcriptome of three mRNAs. (c) shows the k-mers of the read being hashed to vertices in the graph (black circles), thus identifying a subpath. (d) Many k-mers can be skipped because they are covered by the same set of transcripts as a previous k-mer. (e) the set of transcripts compatible with the read is the intersection of the sets of transcripts compatible with its k-mers.

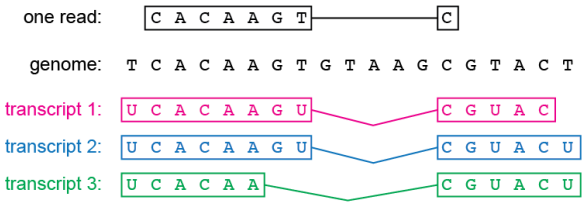
Here's the first main intuition behind kallisto. Each individual k-mer (let's index them with j) in an (error-free) read matches a set of possible transcripts $S_j \subseteq T$. We can find each set S_j quickly by hashing. There's at least one transcript that matches *all* the k-mers in the read. The set S of transcripts compatible with the entire read is the intersection of the compatibilities of its k-mers, $S = \bigcap_j S_j$.

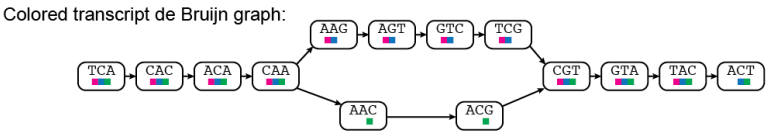
Here's the second main intuition. We can do even better than this, because we have information

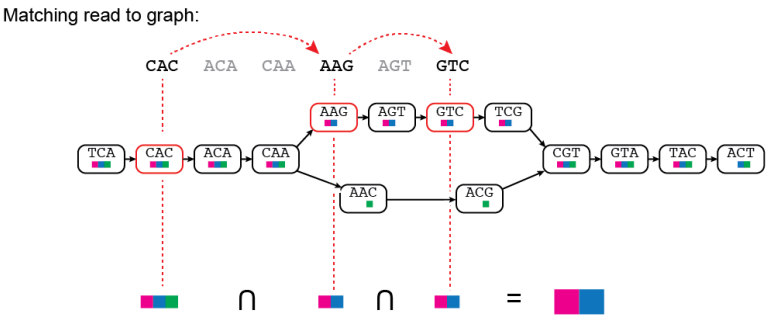
about the expected colinear *order* of the matching k-mers. Suppose we make a graph of the transcriptome with k-mers as its nodes (vertices) and edges connecting directly overlapping k-mers. Transcripts that share overlapping exonic sequence create branches in this graph. Each transcript corresponds to a specific subpath through the graph. Each vertex (each k-mer) has a set of transcript paths that go through it. Adjacent vertices tend to share the same set of paths, until a transcript stops or starts, or the graph branches. We can identify *contigs* in the graph that are covered by the same set of transcripts. We only need the intersection of the sets S_j, so once we've seen one k-mer map to a contig, we don't have to look at any more of the vertices in that contig, and we can skip to a k-mer in the next possible contig(s). This skipping saves many k-mer lookups.

kallisto expects to be operating in a regime of high sequencing accuracy, because it does exact-match hashing of its k-mers. The key parameter is the k-mer length k, which must be short enough that most k-mers are error-free, but long enough that error-free k-mers map only where they're supposed to, and k-mers containing any base calling errors don't map to the transcriptome at all.

Here's a version of the kallisto paper's figure 1 with a toy sequence as an example, so you can more easily follow the operations on example k-mers:







k-mers

A k-mer is just a contiguous subsequence of length k. For example, if we have a read GGAACTGAGGT and $k = 4$, the k-mers on the top strand are GGAA, GAAC, AACT, ACTG, CTGA, TGAG, GAGG, and AGGT.

hashing

A **hash table** is a method for rapid indexing and looking up information on some *key*. Here, our keys are the k-mers. The hash table is composed of a fixed number of slots, with each slot containing a list of keys and whatever information we need about each key. A **hash function** $h(x)$ maps a key x to an integer index that tells us that key x goes in slot $h(x)$. The basic idea is that if you know what keys you're going to need to look up fast -- the k-mers in the known transcriptome -- you can precompute a hash table for them. Then, for each k-mer r_j in a read, we compute the hash function $h(r_j)$, go immediately to that slot, and find the list of transcripts that have that matching

k-mer.

We generally want the hash table to have a number of slots about on the order of the number of keys we need to store. Then instead of having to search over all N key strings of length K to find a match -- naively an $O(KN)$ operation -- we compute a hash function in $O(K)$ and go straight to the matching slot.

An example of a simple hash function: we could just add up the ASCII values of the letters in a key string, and divide modulo the number of slots:

```
def naive_hash(s, hashsize):
    h = 0
    for i in range(len(s))
        h += ord(s[i])
    return h % hashsize
```

This is a bad hash function that will tend to assign slots nonuniformly, because biases in the composition of the keys will produce biases in the hash values. An ideal hash function assigns its keys uniformly to slots. Rarely, we might have a special problem where we could actually do that exactly (by knowing something about the keys), but more commonly, the best we can do is when a hash function maps its inputs randomly (though deterministically and reproducibily) to slots. The way people think about hash functions is therefore related to how people think about deterministic pseudorandom number generators.

Sometimes in DNA/RNA sequence analysis, we use **direct hashing**, where we represent a fixed-length k-mer string by an integer and use that value directly. For example, we can represent a 6-mer sequence GAATTC in base-4 as 200331, and convert the base-4 to 2048 + 48 + 12 + 1 = 2109, and hash it into one of $4^6 = 4096$ slots 0..4095. Because the number of slots in a direct DNA hash is 4^k, we can run into memory problems as k grows.

the transcriptome de Bruijn

graph (T-DBG)

A key difference between Sailfish and kallisto is that Sailfish processes k-mers independently, whereas kallisto keeps information about the order of the k-mers on the read and the transcriptome. kallisto uses the graph structure we described above in the intuition.

A **de Bruijn graph** is a directed graph in which vertices are k-mers, and edges represent overlaps between the k-mers. A sequence is a path through the graph. If we had a perfect error-free de Bruijn graph (and a single chromosome, and no issues with repeats), we could identify a *Eulerian cycle* that visits every edge exactly once, and we'd recover the complete genome sequence assembly.

kallisto uses a variant called a *colored de Bruijn graph*. Each transcript goes through a subpath of vertices; so at each vertex, we keep track of the set of transcripts that cover us. kallisto calls this set the "k-compatibility class" of each k-mer. (In the kallisto paper, the key line is: "the path cover of the transcriptome induces a k-compatibility class for each k-mer.")

the likelihood

Using k-mer hashing on the transcriptome de Bruijn graph, kallisto collects counts c_e for *equivalence classes* $e \in E$ where the equivalence classes E are all the different sets S of transcripts that a read can be assigned to.

The equivalence classes E are essentially the same as the segments S in this week's pset, the Arc locus. What we *want* are counts assigned to *transcripts* i, but what we *have* are counts assigned to equivalence classes; within each equivalence class, there are several possible transcripts, so the source transcript identity is a hidden variable in our inference.

The kallisto paper states the likelihood as:

$$L(\nu) \propto \prod_{f \in F} \sum_{i \in T} y_{f,i} \frac{\nu_i}{\ell_i}$$

where ν_i are the unknown nucleotide abundances of each transcript i, ℓ_i are the (effective) transcript lengths, the F are observed fragments f (from which we've obtained reads from both ends, or a read from one end randomly), and $y_{f,i}$ is an indicator variable that's 1 if fragment f is compatible with transcript i.

When to use transcript abundance (τ_i) versus nucleotide abundance (ν_i) is confusing. The generative likelihood model assumes that we're sample a transcript with probability ν_i (nucleotide abundance; longer transcripts are more likely to be sampled), followed by sampling a fragment position in that transcript with probability $\frac{1}{\ell_i}$.

The likelihood can be rewritten in terms of counts c_e for each equivalence class:

$$L(\nu) \propto \prod_{e \in E} \left(\sum_{i \in e} \frac{\nu_i}{\ell_i} \right)^{c_e}$$

(I had to stare at this last bit for a long time to convince myself that it's right.)

kallisto then finds a vector ν of proposed nucleotide abundances that maximizes this likelihood using an iterative algorithm called expectation-maximization (EM). We'll see this algorithm -- and implement it -- later in the course. Conceptually it's straightforward:

- if I told you ν, then I could apportion one count for each read to each of its compatible transcripts i with probabilities proportional to their transcript abundances, and thus collect expected counts of reads for each transcript. (Expectation step.)

- If I told you the read count for each

transcript i, then you could renormalize to estimate its relative abundance ν_i. (Maximization step.)

- You don't know either one... but if you start from a random guess at ν, and iterate the expectation and maximization steps, you'll converge to a local optimum for ν.

When kallisto reports results, it renormalizes the nucleotide abundances ν_i to transcript abundances τ_i and multiplies by 10^6, thus reporting transcript abundances in units of TPM (transcripts per million transcripts).

Practicalities with kallisto

You can either install kallisto from a pre-compiled package (easy way) or from its source code.

installing kallisto as a package

Two options; you can either install with the Homebrew package manager:

```
% brew install kallisto
```

Or you can install with Anaconda's `conda`:

```
% conda install kallisto
```

More detailed instructions are here.

installing kallisto from source

Instructions are here.

Here's what I did on my Mac OS/X laptop:

```
# install Homebrew package manager
% /usr/bin/ruby -e "$(curl -fsSL https://raw.githu

# use homebrew to install cmake
% brew install cmake
```

```
# ... and autoconf
% brew install autoconf

# ... and the HDF5 library
% brew install hdf5

# get the kallisto source code, unpack, build, and
% wget https://github.com/pachterlab/kallisto/arch
% tar xzf v0.46.1.tar.gz
% cd kallisto-0.46.1/
% cd ext/htslib
% autoheader
% autoconf
% cd ../..
% mkdir build
% cd build
% cmake ..
% make
% make install
```

Now I have kallisto installed in my `/usr/local/bin/kallisto`.

When I ran it, I got an error:

```
dyld: Library not loaded: @rpath/libhdf5.103.dylib
Referenced from: /usr/local/bin/kallisto
Reason: image not found
Abort trap: 6
```

I think what happened is that kallisto's configuration found the HDF5 library in my Anaconda distribution, so it built that path into kallisto, but the Anaconda libraries are not in my normal library load path. I hacked it into submission by making a symlink:

```
% cd /usr/local/lib
% ln -s ~/anaconda3/lib/libhdf5.103.dylib .
```

Now kallisto ran fine for me.

This is a typical installation procedure for command-line bioinformatics tools. (Including the part where it doesn't work right away!) I use a package manager like Homebrew for things I want to Just Work, but I don't want to think about much. For things that I care about how they work, I'll usually build from source, so I'm sure I have the source code around to study. (I've been studying some of the kallisto code.)

Because I study other people's source code a lot, one tool that I use often is sloccount, which gives me a quick snapshot of how much code there is, and in what languages. kallisto's source directory is about 9K lines of C++ code: that's a nice compact codebase, small enough to study carefully.

running kallisto

Good command line tools will print some self-documentation if you just type their name on the command line with no arguments, or maybe with a `-h` (help) option.

kallisto is typical of a good program:

```
% kallisto
kallisto 0.46.1

Usage: kallisto <CMD> [arguments] ..

Where <CMD> can be one of:

    index         Builds a kallisto index
    quant         Runs the quantification algorithm
    bus           Generate BUS files for single-cell
    pseudo        Runs the pseudoalignment step
    merge         Merges several batch runs
    h5dump        Converts HDF5-formatted results to
    inspect       Inspects and gives information abou
    version       Prints version information
    cite          Prints citation information

Running kallisto <CMD> without arguments prints usage
```

The subcommands we're going to care most about are `index` and `quant`.

Remember that in a Jupyter notebook cell, you can run any shell command by prefixing it with `!`. In my examples here, my commands are prefixed with `%` to indicate a system's command line prompt, and I'm typing my commands at the command line. (Specifically, Mac OS/X Terminal.) But you can just as easily be typing them in Jupyter Notebook: for example, `! kallisto`. Just to be clear, if it's your first time: on the command line, the *system* prints the prompt (`%` or whatever) and *you* type the rest (`kallisto`); in Jupyter

notebook, there's no prompt, the `!` is a "magic" Jupyter command. You type the whole thing, `! kallisto`.

OK, back to kallisto.

Good command line tools also come with a small starting example/tutorial, to help you get started. Such examples are super useful to see immediately what the input needs to look like, and what the output is going to come out like. The kallisto source comes with a `test/` subdirectory which contains an example transcriptome in a compressed FASTA file (`transcripts.fasta.gz`) and example paired-end read data in two compressed FASTQ files (`reads_1.fastq.gz` and `reads_1.fastq.gz`).

First we build a kallisto index of the transcriptome, which we'll call `transcripts.idx`:

```
% kallisto index -i transcripts.idx transcripts.fa
```

Then we can map and quantitate reads against that transcriptome. Here, for paired ended data (more on this in a sec):

```
% kallisto quant -i transcripts.idx -o output reac
```

When we got the screen output from `kallisto index`, it looked like:

```
[build] loading fasta file transcripts.fasta.gz
[build] k-mer length: 31
[build] counting k-mers ... done.
[build] building target de Bruijn graph ...  done
[build] creating equivalence classes ...  done
[build] target de Bruijn graph has 27 contigs and
```

which doesn't tell us a lot, except that it worked, it used 31-mers, and its de Bruijn graph (which you know something about, from lecture) has 27 "contigs" and 22118 vertices. It's stored this information in the new index file `transcripts.idx`.

When we got the screen output from `kallisto quant`, it looked like:

```
[quant] fragment length distribution will be estim
[index] k-mer length: 31
[index] number of targets: 14
[index] number of k-mers: 22,118
[index] number of equivalence classes: 20
[quant] running in paired-end mode
[quant] will process pair 1: reads_1.fastq.gz
                             reads_2.fastq.gz
[quant] finding pseudoalignments for the reads ...
[quant] processed 10,000 reads, 9,413 reads pseudo
[quant] estimated average fragment length: 178.02
[   em] quantifying the abundances ... done
[   em] the Expectation-Maximization algorithm ran
```

You can see kallisto working through the steps of its algorithm: loading its index; pseudoaligning the reads; and doing EM to quantitate each transcript. One useful thing to pay attention to here is

```
[quant] processed 10,000 reads, 9,413 reads pseudo
```

where it's telling you it successfully matched 94% of the reads in this toy case. It's typical in real experiments for maybe 80-90% of your reads will match your transcriptome, because your RNA-seq library has other cruft in it: RNAs from microbes and other contaminants that were in your sample, RNAs you didn't put in your transcriptome (ribosomal RNAs and/or mitochondrial RNAs, or unannotated novel transcripts, for example), and reads that had low base-calling quality and couldn't be mapped. Seeing a low number here might be a heads-up that there's something wrong in your reads or your RNA-seq experiment.

fragment lengths; paired vs single ended

Recall that in an RNA-seq experiment, we typically make a double-stranded cDNA copy of RNAs in our sample, fragment that cDNA into small pieces, then obtain a sequence read from one or both ends of a fragment. When we think about how many different fragments we can get from an RNA of length L, we realize there's an edge effect; for a fragment of length ℓ, there are $L - \ell + 1$ different fragments we can sample, not L of them.

So programs like kallisto calculate their TPM estimates using an **effective transcript length**, corrected for the edge effect caused by the fragment length distribution, not the raw transcript length L. This means that kallisto needs to know the distribution of fragment lengths in your experiment.

If you did **paired-end** sequencing, you (tried to) obtain a read from **both** ends of each fragment. You have two input FASTQ files, in exactly the same order: read i in the first file and read i in the second file are the two ends of the same fragment.

Because individual reads can fail on the sequencer for various reasons, or get filtered out in initial quality control checking of your raw sequencing output (and these are things we're not going into in MCB112!), an RNA-seq analysis pipeline will typically have a step where it carefully constructs the paired read files, for the subset of reads where you did indeed obtain both ends successfully for each fragment.

When kallisto sees paired-ended data (two input FASTQ files), it can and will obtain the fragment length distribution itself -- because it sees where the pairs are mapping on the transcriptome, and it can measure the mean and standard deviation of the distance between them. It assumes that the distribution is a truncated Gaussian; truncated, because it assumes that fragments are at least as long as a read, and no longer than the source transcript.

When you use kallisto to analyze single-ended data (which is the case for this week's homework problem), now it can't estimate the fragment length distribution itself. You have to provide the relevant parameters. If you were doing a real RNA-seq experiment, you will have an idea of what they are, roughly, from how you did your fragmentation. In our simulation in this week's homework, you know these parameters exactly.

kallisto needs to be run with an option `--single` to tell it to analyze in single-ended mode, and then it needs options `-l <mean>` and `-s <sd>` to specify the fragment length distribution, as in:

```
% kallisto quant -i transcripts.idx -o output --si

[quant] fragment length distribution is truncated
[index] k-mer length: 31
[index] number of targets: 14
[index] number of k-mers: 22,118
[index] number of equivalence classes: 20
[quant] running in single-end mode
[quant] will process file 1: reads_1.fastq.gz
[quant] finding pseudoalignments for the reads ...
[quant] processed 10,000 reads, 9,034 reads pseudo
[   em] quantifying the abundances ... done
[   em] the Expectation-Maximization algorithm ran
```

gzip compression

The example transcriptome and the example reads are `.gz` files, which means they are compressed in `gzip` format.

You can look at them with `gunzip -c` (the `-c` option says to put the decompressed data on standard output, as opposed to decompressing the file in place). So for example we can peek at the top of the transcriptome file:

```
% gunzip -c transcripts.fasta.gz  | head -2
>ENST00000513300.5
AGTGCCTTTGGCGCACTGAGGTGCACAGGGTCCCTTAGCCGGGCGC...
```

and the top of a read file:

```
% gunzip -c reads_1.fastq.gz | head -12
@1:NM_014620:16:182
GTTCCGAGCGCTCCGCAGAACAGTCCTCCCTGTAAGAGCCTAACCATTGC
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
@2:NM_014620:1094:172
ATGAAAAAAATTCACGTTAGCACGGTGAACCCCAATTATAACGGAGGGGA
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
@3:NM_022658:294:172
TGTACGGGCCCGGCGGCTCGGCGCCCGGCTTCCAGCACGCTTCGCACCAC
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
```

To decompress the files in place, just leave off the

`-c`. But there's no reason to. Increasingly, bioinformatics command-line tools, including kallisto, are able to read compressed data natively, so you don't have to have uncompressed data lying around.

FASTA format for the transcriptome

FASTA format is a simple and widely used format for DNA, RNA, and protein sequences. Like many bioinformatics formats, it is flexible but poorly standardized; so much so that different documentation on FASTA format is typically inconsistent in subtle ways. The lowest common denominator:

Each sequence has a header line that starts with `>`, followed by the sequence **name** (one "word", whitespace-delimited). The remainder of the header line is an optional free-text **description**. To be maximally safe, don't put any space between the `>` and the name; some programs won't recognize the name if you do.

Lines after the header lines are the **sequence**. To be maximally safe as a FASTA input file, these lines should be short (60-80 character lengths are typical), and contain nothing but legal IUPAC one-letter sequence codes for DNA or protein. It's generally not safe to use the RNA `U` instead of `T`, because many programs won't recognize that U and T are the same, or may not recognize U at all.

An example:

```
>NM_001168316  Description of sequence one.
TTAGTGAGGTTGGGGAGAGATAACGCTGTAAACTTTTATTTTTCAGGAAA
TACAGTCTCCAAGCCTGCTCAGCCAAGAAGGAGCTCACTGTGGGCACCAG
CATTT
>NM_174914 Description of sequence two.
TTAGTGAGGTTGGGGAGAGATAACGCTGTAAACTTTTATTTTTCAGGAAA
TACAGTCTCCAAGCCTGCTCAGCCAAGAAGGAGCTCACTGTGGGCACCAG
CCAATGTGGAGACCTGTGAGCCTGTGTCCGGCCCTGAA
```

You can often recognize the source of a sequence

from the format of its name. In the kallisto example transcriptome, names like `NM_001168316` are recognizable as being NCBI RefSeq identifiers. These identifiers are often directly google-able, or you can go to the NCBI web page and search for them.

FASTQ format for the reads

Here again are the first three reads from `reads_1.fastq`:

```
@1:NM_014620:16:182
GTTCCGAGCGCTCCGCAGAACAGTCCTCCCTGTAAGAGCCTAACCATTGC
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
@2:NM_014620:1094:172
ATGAAAAAAATTCACGTTAGCACGGTGAACCCCAATTATAACGGAGGGGA
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
@3:NM_022658:294:172
TGTACGGGCCCGGCGGCTCGGCGCCCGGCTTCCAGCACGCTTCGCACCAC
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
```

A typical FASTQ file has four lines per read:

- The **name** line starts with `@`, followed by a one-word name. The Illumina Casava pipeline follows a standard naming system that encodes a bunch of information into the read name, including the instrument ID, run number, flowcell ID, lane number, tile number, x/y positions of the cluster on the flowcell -- and to understand what these mean, you have to know something about how Illumina sequencing technology works. In the kallisto example files, the names appear to be `<read_number>:<source_transcript>:<fragment_start>:<fragment_length>` (I'm guessing; haven't confirmed).
- The **sequence** line: this is the read sequence.
- A line with only a `+` on it. (Who knows. Don't ask.)

- The **quality values (QVs)**, same length as the read sequence, one character for each base in the read. I'll define the quality values below.

quality values in FASTQ

A "quality value" (QV) is the scaled log probability of a base-calling error $(1 - \alpha)$, for base calling accuracy α:

$$\mathrm{QV} = \mathrm{round}(-10 \log_{10} P(1 - \alpha))$$

For example, if the estimated error probability is 0.01, this is a Q20 base. If the estimated error probability is 10^{-4}, this is a Q40 base.

The bases in a typical Illumina raw read are about Q30 [Lee et al. 2016]. QV's are encoded as ASCII characters. Sanger format encodes QVs 0..93 as ASCII 33..126 (sometimes called "base 33" QV encoding). Historically, there was confusion over which encoding scheme is to be used, with Illumina software using a different scheme, but Illumina's CASAVA pipeline finally adopted Sanger format as of release 1.8 in 2011.

For example, a Q20 base is encoded as ASCII character 53, which is '5'; a Q30 base as '?'; a Q40 base as 'I'.

Practicalities with Python in the pset

numpy

Some of you are already familiar with Python modules, but we're assuming you're not. Last week, we only used basic Python. Now we'll use our first module: NumPy, numerical Python. NumPy provides tools for arrays, vectors, linear algebra, and random number generation and sampling. A good tutorial is here, but you don't

need to go through it yet to get started on the homework.

You "import" modules you want to use at the start of your script: `import numpy`, for example. Optionally, you can give the module an alias, and NumPy is commonly aliased to `np`:

```
import numpy as np
```

Now I can call NumPy functions by prefixing them with `np.`: for example,

```
np.log10(100)
```

prints 2.0. Without the alias, I would've used the full module name, `numpy.log10(100)`.

creating numpy arrays

`np.array()` takes a Python list as an argument and returns a NumPy array. For example, suppose I want to simulate rolling a loaded die; I'll want to define a probability vector for my dice:

```
p = np.array( [ 0.1, 0.1, 0.1, 0.1, 0.1, 0.5 ] )
```

This extends to creating 2D (or N-dimensional) arrays from lists of lists. Suppose I want to parameterize a fair die and a loaded one:

```
p =  np.array([ [ 1/6, 1/6, 1/6, 1/6, 1/6, 1/6 ],
                [ 0.1, 0.1, 0.1, 0.1, 0.1, 0.5 ] ])
```

Now `p[0]` is a probability vector for my fair die, and `p[1]` is my loaded die.

NumPy has a lot of different methods for initializing arrays so there are other ways to do this too. One useful method is `np.zeros()`, which creates an array of specific dimensions that's initialized to all zeros. `np.zeros(3)` returns `[0., 0., 0.]`, a 1x3 vector. `np.zeros(2,3)` returns `[ [0., 0., 0.], [0., 0., 0.]]`, a 2x3 array. `np.ones()`, unsurprisingly, does the same thing but

with 1's.

numpy math on vectors and arrays

NumPy provides methods for doing arithmetic, especially with vectors and arrays. For an introductory example, here's how we can normalize a vector so it sums to one ... for example, if we started with some observed counts of die rolls, and we wanted to convert counts to frequencies:

```
p   = np.array( [ 10, 10, 10, 10, 10, 50 ] )
sum = np.sum(p)
p   = np.divide(p, sum)
```

random number generation

To simulate data sets, we need to do random sampling. The most basic routine is a *random number generator* (RNG) that generates a uniformly distributed sample on some interval. It's very common for RNGs to provide a floating point value u, uniformly distributed on the interval $0 \leq u < 1$: u can be exactly zero, but it isn't exactly one. This is what NumPy provides:

```
u = np.random.rand()      # Sample a single random
V = np.random.rand(10)    # Sample a vector of 10
A = np.random.rand(2,10)  # Sample a 2x10 array of
```

There's a lot to say about random number generators, but two of them are super important.

The first thing is that almost all random number generators (RNGs; including the "Mersenne Twister" generator that NumPy uses) are actually *pseudorandom*. The sequence of numbers they generate appears statistically random, but if you start the generator in the same internal state, you get exactly the same sequence.

Because cryptography algorithms often depend on random number generation, cryptography

aficionados go all haywire about pseudorandomness. If you know the RNG algorithm and you can deduce its internal state, you can completely predict everything the RNG will generate. This is less than optimal, as you can imagine, if that sequence was supposed to be a secret.

The internal state of an RNG is initialized by a single number called the **seed**. If you initialize the RNG with the same seed, it generates a reproducible sequence of numbers. You can try that in NumPy for yourself:

```
np.random.seed(42)
for i in range(10):
   print(np.random.rand())
```

Execute that code (or Jupyter notebook cell) several times; you'll see the same sequence of numbers.

Which brings us to **the second thing**. Sometimes you want to get a reproducible result from your simulation experiment, but sometimes you want to run independent replicates. If you want a reproducible result, you run with a *fixed* RNG seed. If you want independent replicates, you run each replicate with a *different* RNG seed.

If you *don't* call `np.random.seed()`, the NumPy RNG will select a random seed, so you get a different sequence of numbers every time you run your program. (Or, if you call `np.random.seed(None)`.)

Think about what NumPy has to do. To get a different RNG seed for each run, you have a chicken and the egg problem - you need an RNG to give you a random seed to start your RNG. There are a variety of strategies to generate pseudorandom seeds, most of them bad. You need to be aware of their shortcomings.

If it has no better option, NumPy will use the system clock time, which might be in units of seconds; this is a very common strategy for

getting a quasirandom seed. But this can mean that if you start many instances of your program quickly (like, firing off 100 replicate runs from a script), ones that start in the same time interval will get the same RNG seed and produce identical results. Not what you want.

For this reason, `numpy.random` will try to use other, better sources of randomness to initialize the seed, if the operating system provides them. (Linux and OS/X provide a cryptographically secure hardware RNG, `/dev/urandom`.) But now this sort of thing raises other issues, from the standpoint of reproducibility of scientific results: now your results depend in detail on the operating system your script is run on. "It works fine on my OS/X machine", but the NumPy docs make no guarantees; maybe on some systems I can't get away with starting a bunch of replicates in a narrow time interval. I also don't know that the same seed will produce the same random number sequence on different platforms. This can become a nettlesome issue.

random sampling

On the sturdy back of uniform 0..1 sampling with `np.random.rand()`, NumPy builds a variety of random sampling methods. Ones that are most useful to start with include:

```
n = np.random.randint(1, 7)
c = np.random.choice(list('ACGT'))

p = [0.1,0.1,0.7,0.1])
c = np.random.choice(list('ACGT'), p=p)

A = np.random.choice(list('ACGT'), 100)
S = ''.join(np.random.choice(list('ACGT'), 100))

u = np.random.normal(mean, stdev)

A = list('ABCDEFGHIJ')
np.random.shuffle(A)
```

circular permutation as an excuse to mention integer

arithmetic

The Arc locus in the homework is a circle composed of 10 segments. When you're indexing on a circle, you need to roll around from the end (10) back to the start (1): i.e., the Arc10 transcript is JAB, segments 10-1-2. To do stuff like this you'll want to know about the integer modulus operator, `%`, which means `remainder`. For example:

```
for j in range(100):
    i = j % 10
    print(i)
```

While we're at it... sometimes you need to be careful in code to distinguish *integer* computations from *floating point* computations (the computer's approximate representation of real numbers). These are different data types. One place where you can get burned is in the difference between integer division and floating point division. You might think 5/3 is a real number 1.6666... (infinitely) but it's 1 in integer division, and it's rounded off to some loss of precision in floating point.

In Python3, division *always* results in a float return value, even with integer arguments: `5/2 = 2.5` and `5.0/2.0 = 2.5`.

If you want integer division, you can use the `//` operator, which is *floor division* for floats: return the largest integer not greater than the result of the computation. `5//2` is 2, and `-5//2` is -3.

Alternatively you can convert floats to ints with `int()`, but watch out, this is not the same as floor division. `int(5/2)` is 2, but `int(-5/2)` is -2, not -3: `int()` simply drops any precision after the decimal point.

(This is a good rabbit hole to dive into if you want to be terrified about how anything ever works at all. For example, some Python documentation is incorrect about how floor division works. And

Python2 behaves *differently* with respect to its implementation of the `/` division operator! Division may not give you the same answer in Python2 versus Python3 code. Here's Python's designer, Guido van Rossum, apologizing for the mess. Who cares about these details, you say? Well, for one: the European Space Agency's $7 billion Ariane 5 rocket exploded on its first launch because someone erroneously converted a 64-bit float to a 16-bit unsigned integer in the wrong place.)

manipulating and transforming strings

Data types in Python are *objects*, and objects have *methods* that you can call on them to manipulate their contents. You get used to the idea of even what seem like elemental data types being objects with methods. (I'm a C programmer; for me, it takes a *lot* of getting used to.) A good and useful place to practice is with strings. We've already seen the `<str>.format()` method, in learning how to print stuff, and the '.split()' method for splitting a line into data fields, but you may not have been thinking about these in terms of methods that operate on a data object. Here's some other examples:

```
S  = '{:.2f}'.format(1.2345)          # S = '1.2
S  = 'acgt'.upper()                   # S = 'ACG
S  = ','.join(['abc', 'def', 'ghi'])  # S = 'abc
```

Here's a more complex example that will be useful in this week's homework (though there are a jillion other ways to do it too). `<str>.translate(tbl)` translates each character in `<str>` to another character, according to a so-called "translation table" `<tbl>`. You create a translation table using `str.maketrans(from, to)`, where (a detail!) `str` is literal, not a variable (`str.maketrans` is a so-called static method). For example, this little useful incantation:

```
S = 'AAAAAAAAAAAAA'.translate(str.maketrans('ACGT
```

returns 'TTTTTTTTTTTTT', having converted A's to T's (and C to G, G to C, T to A).

further reading

- Bo Li, Victor Ruotti, Ron M. Stewart, James A. Thomson, Colin N. Dewey; RNA-Seq gene expression estimation with read mapping uncertainty Bioinformatics 26:493 (2010)
- Nicolas Bray, Harold Pimentel, Pall Melsted, Lior Pachter; Near-optimal probabilistic RNA-seq quantification; Nature Biotechnology 34:525 (2016)
