THE LOCH NESS MONSTER

This simulation of the Master Equation is based on the observation that the creation and annihilation operators, $a^\dagger$ and a are really matrices. It might help with understanding what is going on in Behr's Molecular Time Machine [priv. comm.], though unlike that approach, here we sidestep the combinatoric complications and simply calculate the result directly.

The direct technique only works without modification for a single species, but supports any number of reactions. It gets more expensive with large numbers of particles because computing the matrix exponential is $\mathcal{O}(n^3)$ [1] in the size of the matrix and we need an ever larger one for more particles.

We start with the Master Equation,

$$\frac{d\left|\psi(t)\right\rangle}{dt} = H\left|\psi(t)\right\rangle \tag{1}$$

and following [2] write the reaction Hamiltonian in terms of creation, $a^\dagger$ and annihilation a operators as,

$$H = \sum_{\tau\in T} r_\tau\left(a^{\dagger t_\tau} - a^{\dagger s_\tau}\right)a^{s_\tau} \tag{2}$$

where T is our set of reactions, and for each of them r is the rate at which it happens, t is the number of particles created and s is the number destroyed.

The solution of equation 1 is,

$$\left|\psi(t)\right\rangle = e^{tH}\left|\psi(0)\right\rangle \tag{3}$$

and we want to have a way of evaluating that exponential. We expect to have a matrix that can be multiplied with a probability distribution (the likelihood to have various numbers of particles) to get the distribution at some point in the future.

So start by defining our operators. Spelling it out, the creation operator, $a^\dagger$ looks like this,

$$a^\dagger = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \cdots \\ 1 & 0 & 0 & 0 & 0 & \\ 0 & 1 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 1 & 0 & 0 & \\ 0 & 0 & 0 & 1 & 0 & \ddots \\ \vdots & & \vdots & & \ddots & \ddots \end{bmatrix} \tag{4}$$

It is easy to satisfy oneself (or prove by induction) that this infinite matrix does what is meant by the definition,

$$a^\dagger\left|n\right\rangle = \left|n+1\right\rangle \tag{5}$$

This matrix can be generated easily from a sparse association list, for arbitrary size – since we obviously cannot generate infinite matrices as such. It is transformed to a dense matrix simply because the matrix exponential function *expm* that we will eventually use expects a dense matrix. It is easy to imagine one that operates on sparse matrices, does things in parallel, uses the GPU, and so forth but premature optimisation is the enemy of getting things done.

```
create :: Int → Int → Matrix Double
create n p = LD.toDense $ ((n-1, n-1), 0):m
  where m = [ ((j+p, j), 1) | j ← [0..(n-p-1)]]
```

The first argument `n` to `create` is the size of the matrix to produce, and the second one, `p`, is the power. We can calculate powers of these matrices directly instead of multiplying out, so this is an optimisation. So this function produces a matrix of size `n` for $a^{\dagger p}$. The association list is prepended with a zero entry in the bottom right corner to establish the intended size of the matrix.

Date: November 29, 2014.

The annihilation operator a is very slightly more complicated,

$$a\,|n\rangle = \begin{cases} n\,|n-1\rangle & n > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

This can be explicitly represented as an infinite matrix in a similar way,

$$a = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 2 & 0 & 0 & \\ 0 & 0 & 0 & 3 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 4 & \\ 0 & 0 & 0 & 0 & 0 & \ddots \\ \vdots & & \vdots & & \ddots & \ddots \end{bmatrix} \tag{7}$$

To do this in code we need either to make a bunch of matrix multiplications, or to implement a falling factorial and construct a sparse matrix as we did above. We do the latter, and define a falling factorial operator,

```
(!.) :: Int → Int → Int
n !. k = product [(n-k+1)..n]
```

and then a^p is implemented in code,

```
annihilate :: Int → Int → Matrix Double
annihilate n p = LD.toDense $ ((n-1, n-1), 0):m
  where m = [ ((j, j+p), fromIntegral (j+p !. p)) | j ← [0..(n-p-1)] ]
```

where the arguments have the same meaning as with `create` – that is `n` is the size of the matrix to produce and `p` is the power that we can calculate directly thanks to our falling factorial operator.

With these in hand, we can define reactions as they appear in the Hamiltonian above, with a certain matrix size as before, `n`, and a certain number of instances of particles that they create `t` or destroy `s` at a certain rate, `r`,

```
reaction :: Int → Int → Int → Double → Matrix Double
reaction n t s r = scale r (ct × as - cs × as)
  where
    ct = create n t
    cs = create n s
    as = annihilate n s
```

Now we can make a function to create Hamiltonians of size n for a list of reaction specifications. Reaction specifications are of the form `(t, s, r)` or in other words the parameters that we need to create an individual reaction.

```
hamiltonian :: Int → [(Int, Int, Double)] → Matrix Double
hamiltonian n rs = foldl1 (+) $ map(λ(t, s, r) → reaction n t s r) rs
```

In words, we create a reaction for each specification in the list, and then add them all together.

It is now simple to write an evolution function that, given a time, initial state and a hamiltonian, returns a new state,

```
evolve :: Double → Matrix Double → Matrix Double → Matrix Double
evolve t h x = (expm $ scale t h) × x
```

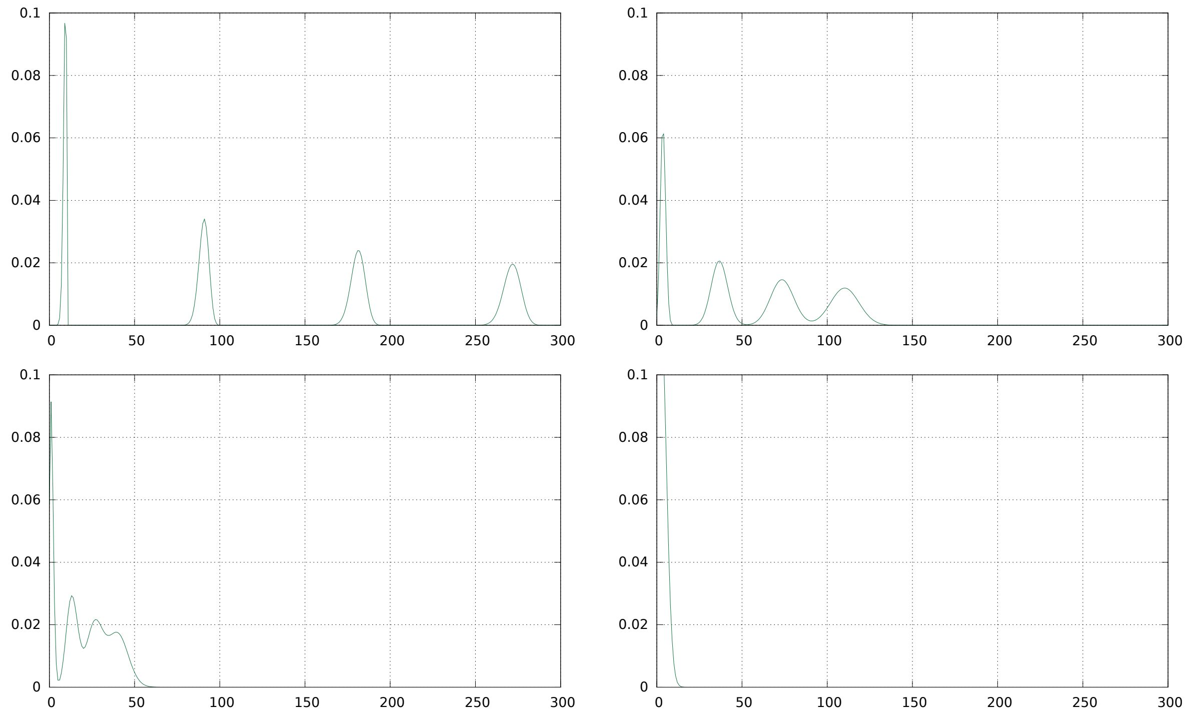

FIGURE 1. Probability distribution evolution for the dissipation reaction. Top row is $t = 0.1$ and $t = 1$, bottom row is $t = 2$ and $t = 4$. The initial distribution is 25% probability for 10, 100, 200 or 300 particles, and 0 otherwise.

Th `evolve` function is almost verbatim an implementation of equation 3, the solution to the Master Equation.

Finally a utility function that does no particular computation but just conducts the evolution `n` times for an interval of `dt` and saves each step to a file on disk,

```
timeseries :: FilePath → Int → Double →
              Matrix Double → Matrix Double → IO ()
timeseries base n dt h x = mapM_ (λk → dump k (fromIntegral k*dt)) [1..n]
  where
    filename k = printf "%s_%04d.dat" base k
    dump k t   =
      saveMatrix (filename k) "%f" $ evolve t h x
```

Again, much room for optimisation here, especially since each step can be carried out in parallel.

Example: Dissipation. Here is a simple driver program to compute the dissipation reaction,

$$A \rightarrow \emptyset \tag{8}$$

where it is fed with an initial probability distribution of equal probability to have 10, 100, 200 or 300 particles. This particular example is chosen because it is the same one that was demonstrated using the MTM and shows that, in this case, the same results can be obtained directly. This should not be surprising, after all the example is one of the simplest possible so no conclusions can be drawn based on these results about how the MTM and the direct methods would compare on more complicated problems.

```
main :: IO ()
main =
  let size = 330
      h = hamiltonian size [(0, 1, 1)]
      x = trans $ LD.toDense [((0,10), 0.25)
                             , ((0,100), 0.25)
```

```
                              , ((0,200), 0.25)
                              , ((0,300), 0.25)
                              ,((0,size-1), 0)
                              ]
  in timeseries "dissipation" 100 0.1 h x
```

The results are plotted in figure 1, and show the familiar Loch Ness Monster curve. It takes about 15 seconds to run for 100 time steps on a reasonably modern laptop computer.

Notes on the method

Size of the matrices. The matrices must be bigger than the largest number of particles expected. Just how much bigger is an interesting question. Here, 10% bigger was arbitrarily chosen, but a more rigorous error bound should be easily obtainable. This bound should hold regardless of the approximation method.

Approximating exponentials. When working with floating point numbers, using a Taylor expansion to approximate a matrix exponential is known to be a bad idea. It is inefficient and numerically unstable. Using arbitrary precision numbers it is stable, but still inefficient. The method used here is as implemented in the *expm* function with Padé approximants and scaling and squaring, which is the "standard" method. It is still expensive, though, and where the task is to multiply a matrix exponential by a vector there are better ones – Krylov iteration and Leja point iteration.

Scaling to multiple reactions. The method implemented here, though it is not very elegant, scales immediately to any number of reactions – the only added cost is in the initial computation of the Hamiltonian which consists only of adding and subtracting matrices and in any case is only done once.

Scaling to multiple species. This is more difficult because it would involve three dimensional matrices, with an extra axis for species. Some work on the data-structures involved would be necessary, as they would in any case for more efficient calculation of the exponentials themselves.

Parallelism. There are numerous places here where parallelism can be exploited. The methods for calculating the exponential could be made to run on multiple cores and to exploit GPU hardware. Because the solution is exact, time-steps can also be calculated independently on different computers in a cluster and the results collated.

Frequency Domain. There is, in this case mostly due to the initial conditions, the strong appearance of periodic behaviour. This suggests it might be profitable to make a fourier transform of the probability distribution. How does such a transform affect the operators? Can it be used to make a diagonal Hamiltonian in the frequency basis? If so, computing the exponential is much simpler.

References

[1] C. Moler and C. Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45(1):3–49, January 2003.

[2] John C. Baez and Jacob Biamonte. A course on quantum techniques for stochastic mechanics. *arXiv:1209.3632 [math-ph, physics:quant-ph]*, September 2012. arXiv: 1209.3632.