THE LOCH NESS MONSTER

This implementation of the Master Equation is a bit naive and is based on the observation that the creation and annihilation operators, $a^\dagger$ and a are really matrices. We start with the Master Equation,

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

and the reaction Hamiltonian written in t.erms of creation, $a^\dagger$ and annihilation a operators as,

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

where T is our set of reactions.

The solution of equation 1 is,

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

and we want to have a way of evaluating that exponential.

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

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

This can be generated easily from a sparse association list, for arbitrary size. 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 to `create` is the size of the matrix to produce, and the second one, as an optimisation, is the power. So this function produces a matrix of size n for $a^{\dagger p}$.

Whereas the action of the creation operator is,

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

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,

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

Date: November 28, 2014.

To implement 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.

```
(!.) :: 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)] ]
```

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
```

For convenience, we can make a function to create for us Hamiltonians, of size n for a list of reaction specifications,

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

and 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
```

This last function implements 3, the solution to the Master Equation.

Finally a utility function that conducts the evolution and saves each step to 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.

Here is a simple driver program to compute the dissipation reaction,

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

where it is fed with an initial probability distribution of equal probability to have 10, 100, 200 or 300 particles.

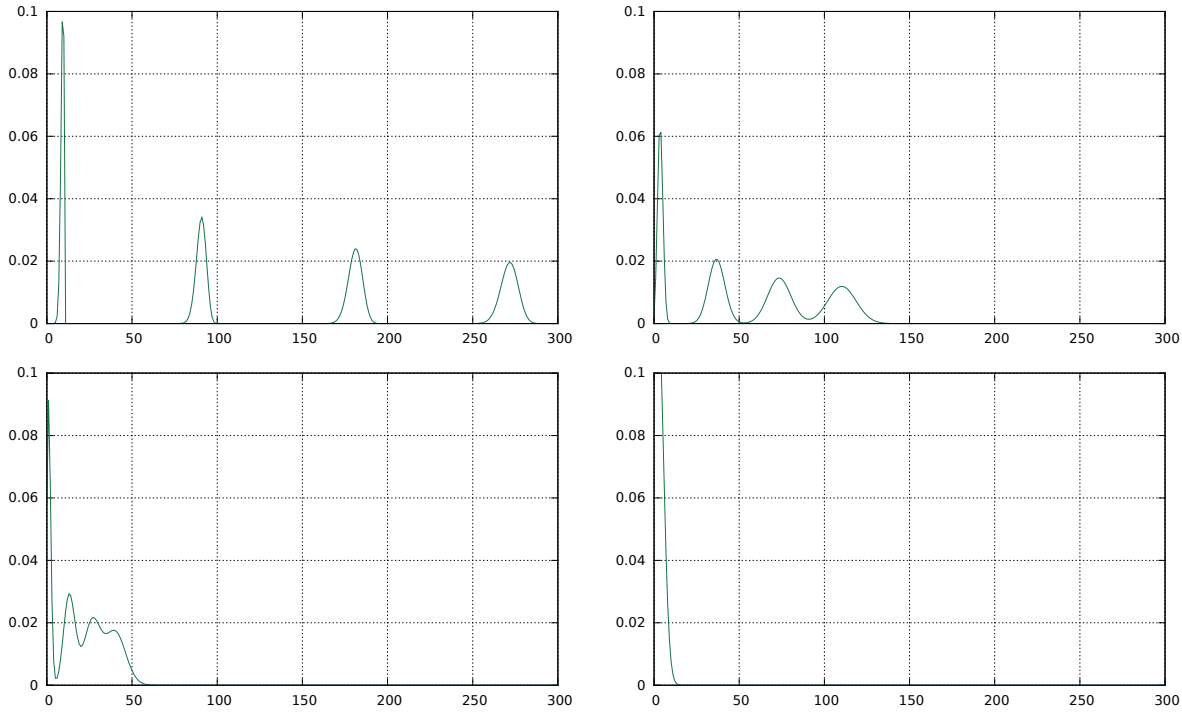

FIGURE 1. Probability distribution evolution for the dissipation reaction. Top row is $t = 0.1$ and $t = 1$, bottom row is $t = 2$ and $t = 4$. The initial distribution is 25% probability for 10, 100, 200 or 300 particles, and 0 otherwise.

```
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. Performance would also degrade at least polynomially in the number of species.

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.