Numerical Quantum Mechanics in Haskell

Andrew Knapp

September 8, 2016

1 Motivation

Since the 1920's, physicists have been thrust into the realm of quantum mechanics. Unfortunately, this means that analytical solutions are much more difficult to come by than in the classical realm. In most introductory courses, the second or third example system results in an (unsolveable) transcendental equation. Hence physicists and engineers that use quantum mechanics to solve problems have embraced numerical computing wholeheartedly. There are still many difficulties, however, since the dimension of the state space grows exponentially with the number of particles. To tame the explosion of computational complexity, physicists look at simplified systems. This article examines one such system - a group of molecules whose resonant frequencies are distributed in a given manner interacting with a laser. We will observe a phenomena known as a photon echo, where a quantum system absorbs a pulse from a laser, and reemits it at a later time.

1.0.1 Assumed Knowledge

This article assumes a working knowledge of the mathematics and notation of introductory quantum mechanics, conceptual understanding of the density matrix, and a working understanding of the Runge-Kutta method. It also assumes familiarity with Haskell's `lens` library.

2 Physics Background

2.0.1 The System of Interest

We are interested in the response of a system of different types of molecules to a laser pulse. These molecules have different resonant frequencies, and their number density as a function of frequency is given by $g(\omega)$. We wish to produce a photon echo, where the system absorbs a pulse of light, and reemits it at a later time. These echoes have been used for optical memory, spectroscopy, and studying the evolution of intermolecular reactions.

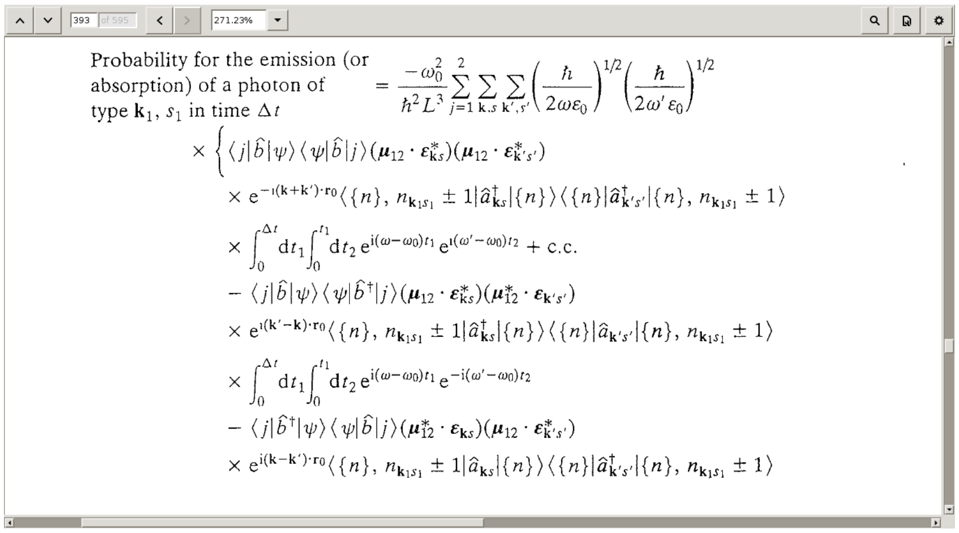
Sadly, this system is far too complex to solve analytically, or even numerically without making some approximations. We will model this system as a collection of two-level atoms that have only excited and ground states. Even if we do this, the system is still far too ugly to work with in an article like this:

To further simplify things, we'll restrict the system to one dimension, assume the atoms are far enough apart that their interaction is negligible, and ignore the interaction of the atoms with the surface they are resting on. This surface is of critical importance in real calculations of this kind, but that involves dragging in Maxwell's equations. For our purposes, we'll only need one set of coupled ordinary differential equations.

2.0.2 The Maxwell-Bloch Equations

Our system's wavefunction can be written in the following form:

$$\psi = c_g \psi_g + c_e \psi_e$$
$$|c_g|^2 + |c_e|^2 = 1$$


393 of 595 271.23%

$$\text{Probability for the emission (or absorption) of a photon of type } \mathbf{k}_1, s_1 \text{ in time } \Delta t = \frac{-\omega_0^2}{\hbar^2 L^3} \sum_{j=1}^{2} \sum_{\mathbf{k},s} \sum_{\mathbf{k}',s'} \left(\frac{\hbar}{2\omega\varepsilon_0}\right)^{1/2} \left(\frac{\hbar}{2\omega'\varepsilon_0}\right)^{1/2}$$

$$\begin{aligned}
&\times \Big\{ \langle j|\hat{b}|\psi\rangle\langle\psi|\hat{b}|j\rangle(\boldsymbol{\mu}_{12}\cdot\boldsymbol{\varepsilon}^*_{\mathbf{k}s})(\boldsymbol{\mu}_{12}\cdot\boldsymbol{\varepsilon}^*_{\mathbf{k}'s'}) \\
&\quad \times e^{-i(\mathbf{k}+\mathbf{k}')\cdot\mathbf{r}_0}\langle\{n\}, n_{\mathbf{k}_1 s_1} \pm 1|\hat{a}^\dagger_{\mathbf{k}s}|\{n\}\rangle\langle\{n\}|\hat{a}^\dagger_{\mathbf{k}'s'}|\{n\}, n_{\mathbf{k}_1 s_1} \pm 1\rangle \\
&\quad \times \int_0^{\Delta t} dt_1 \int_0^{t_1} dt_2\, e^{i(\omega-\omega_0)t_1} e^{i(\omega'-\omega_0)t_2} + \text{c.c.} \\
&\quad - \langle j|\hat{b}|\psi\rangle\langle\psi|\hat{b}^\dagger|j\rangle(\boldsymbol{\mu}_{12}\cdot\boldsymbol{\varepsilon}^*_{\mathbf{k}s})(\boldsymbol{\mu}^*_{12}\cdot\boldsymbol{\varepsilon}_{\mathbf{k}'s'}) \\
&\quad \times e^{i(\mathbf{k}'-\mathbf{k})\cdot\mathbf{r}_0}\langle\{n\}, n_{\mathbf{k}_1 s_1} \pm 1|\hat{a}^\dagger_{\mathbf{k}s}|\{n\}\rangle\langle\{n\}|\hat{a}_{\mathbf{k}'s'}|\{n\}, n_{\mathbf{k}_1 s_1} \pm 1\rangle \\
&\quad \times \int_0^{\Delta t} dt_1 \int_0^{t_1} dt_2\, e^{i(\omega-\omega_0)t_1} e^{-i(\omega'-\omega_0)t_2} \\
&\quad - \langle j|\hat{b}^\dagger|\psi\rangle\langle\psi|\hat{b}|j\rangle(\boldsymbol{\mu}^*_{12}\cdot\boldsymbol{\varepsilon}_{\mathbf{k}s})(\boldsymbol{\mu}_{12}\cdot\boldsymbol{\varepsilon}^*_{\mathbf{k}'s'}) \\
&\quad \times e^{i(\mathbf{k}-\mathbf{k}')\cdot\mathbf{r}_0}\langle\{n\}, n_{\mathbf{k}_1 s_1} \pm 1|\hat{a}_{\mathbf{k}s}|\{n\}\rangle\langle\{n\}|\hat{a}^\dagger_{\mathbf{k}'s'}|\{n\}, n_{\mathbf{k}_1 s_1} \pm 1\rangle
\end{aligned}$$

Figure 1: Better eat your Wheaties before working with this one; it goes onto the next page. (From L. Mandel & E. Wolf's *Optical Coherence and Quantum Optics*)

The subscripted g represents the ground state, while the subscript e represents the excited state. The density matrix is then given by

$$\rho = \begin{pmatrix} \rho_{gg} & \rho_{ge} \\ \rho_{eg} & \rho_{ee} \end{pmatrix} = \begin{pmatrix} c_g c_g^* & c_g c_e^* \\ c_e c_g^* & c_e c_e^* \end{pmatrix}$$

From this starting point and the Jaynes-Cummings model, the Maxwell-Bloch equations can be derived

$$\begin{aligned}
\frac{d\rho_{gg}}{dt} &= \gamma\rho_{ee} + \frac{i}{2}(\Omega^*\bar{\rho}_{eg} - \Omega\bar{\rho}_{ge}) \\
\frac{d\rho_{ee}}{dt} &= -\gamma\rho_{ee} + \frac{i}{2}(\Omega\bar{\rho}_{ge} - \Omega^*\bar{\rho}_{eg}) \\
\frac{d\bar{\rho}_{ge}}{dt} &= -\left(\frac{\gamma}{2} + i\delta\right)\bar{\rho}_{ge} + \frac{i}{2}\Omega^*(\rho_{ee} - \rho_{gg}) \\
\frac{d\bar{\rho}_{eg}}{dt} &= -\left(\frac{\gamma}{2} - i\delta\right)\bar{\rho}_{eg} + \frac{i}{2}\Omega^*(\rho_{gg} - \rho_{ee})
\end{aligned}$$

γ is a decay coefficient that determines how fast the system spontaneously radiates photons. More sophisticated models include two gammas, where one represents emission and the other represents dephasing. Since any decay will weaken the photon echo, we'll set $\gamma = 0$. We can then leave out the equation updating ρ_{eg}. Ω is the so-called generalized Rabi frequency that relates the frequency of the laser pulse with detuning δ from the resonant frequency of the atom to the atom's frequency of spontaneously emitted radiation.

The dynamics of the 1D Maxwell-Bloch equations can be visualized as taking place on the unit circle (the above picture is for the 2D case). Since $|c_g|^2 + |c_e|^2 = 1$, the vector represented by these two components is of unit length, and any evolution will be a simple rotation. When the atom is excited by a laser pulse, this is exactly what happens. I could say a lot more about this, but the most important mathematical fact about this evolution is that the speed of rotation depends on the atom's resonant frequency and the pulse's deviation from it.

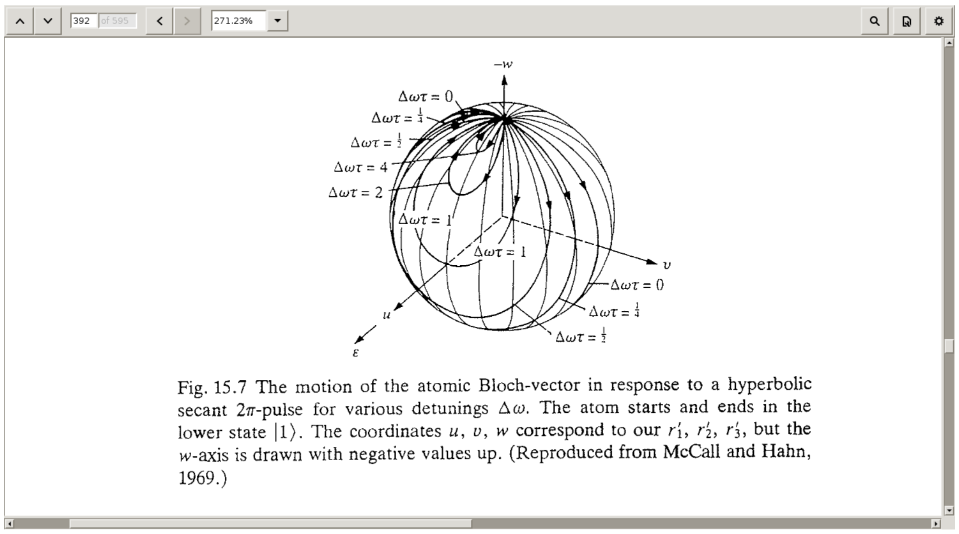


Fig. 15.7 The motion of the atomic Bloch-vector in response to a hyperbolic secant 2π-pulse for various detunings $\Delta\omega$. The atom starts and ends in the lower state $|1\rangle$. The coordinates u, v, w correspond to our r_1', r_2', r_3', but the w-axis is drawn with negative values up. (Reproduced from McCall and Hahn, 1969.)

Figure 2: The Bloch Sphere for the 2D Maxwell-Bloch Equations (From L. Allen & J.H. Eberly's *Optical Resonance and Two-Level Atoms*)

2.0.3 Photonic Echoes and π-pulses

Suppose we have a pulse at frequency ω with envelope $E(t)$. As time passes, the pulse will move the state vector around the circle. If the area under the envelope is an integer multiple of π and a normalization factor, the state vector is returned to its starting point, which is usually the ground state. However, half-integer multiples of π flip the system's state vector 180° at time τ - the system is now fully excited. If we have a system of different kinds of atoms with different resonant frequencies, and zap it with a $\pi/2$-pulse, there will be a τ where every type of atom is excited at the same time. This is known as *coherent rephasing*. We can visualize this as a group of runners moving on a circular track of unit length. If their velocities are all half-integers, they will "echo" their starting configuration on the opposite side of the track.

In our system, this realignment in the excited state causes a spontaneous emission of a pulse with the same shape as the original. In the real world, the intensity will be much less than the original, due to dephasing, spontaneous emission, environmental effects, etc. In our very simple model, however, the echo will be an almost exact copy of the original pulse. To observe this, we will look at the net dipole moment of the system, because it allows us to observe the emissions of the system without having to set up a grid to use Maxwell's equations.

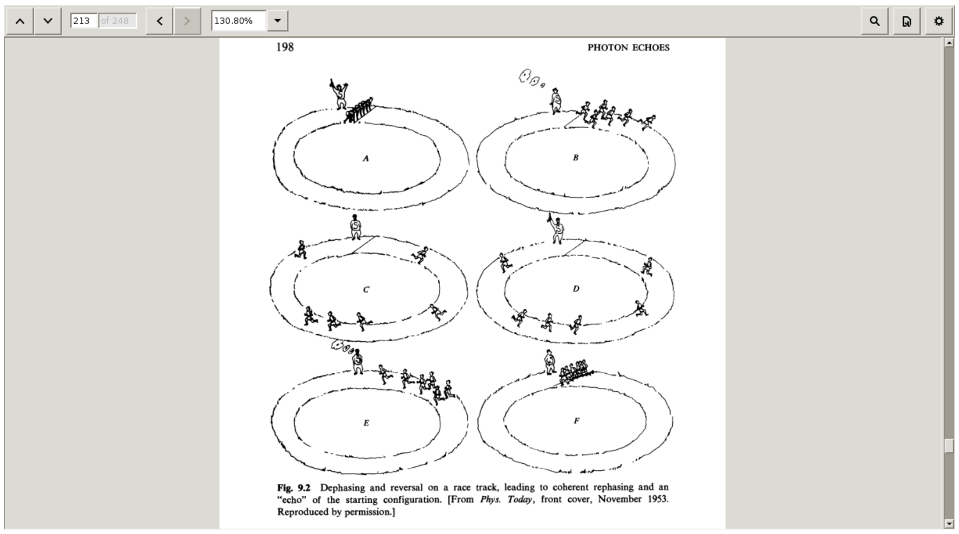
3 Implementation in Haskell

First, we need some imports and language pragmas.

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# LANGUAGE BangPatterns #-}

module Main where

import Prelude hiding (Real)
```



198 PHOTON ECHOES

Fig. 9.2 Dephasing and reversal on a race track, leading to coherent rephasing and an "echo" of the starting configuration. [From *Phys. Today*, front cover, November 1953. Reproduced by permission.]

Figure 3: Dephasing and Rephasing on the Bloch Sphere

```
import Control.Lens
import Control.Parallel.Strategies

import Data.Complex
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Vector.Unboxed as U

import System.Environment
import System.IO
```

We'll now define a data structure to hold some configuration variables that the user might want to alter. This code doesn't do so, but these values could be used in a config file to avoid recompilation. For a larger project, we could use a reader monad transformer, but for code this short, we're better off keeping things pure. We'll use lenses, though.

```
type Real = Double

data SimulationConfig = SimulationConfig
    {
      _d_omega :: !Real  -- ^ the spread from the central resonant frequency
    , _dp      :: !Real  -- ^ the transition dipole moment
    , _dt      :: !Real  -- ^ the timestep for the system
    , _e0      :: !Real  -- ^ pulse envelope magnitude (electric field strength)
    , _nt      :: !Int   -- ^ the number of timesteps
    , _nw      :: !Int   -- ^ the number of different frequencies
    , _omega0  :: !Real  -- ^ the resonant frequency
    , _tau     :: !Real  -- ^ the pulse duration
    } deriving (Show, Eq)

makeLenses ''SimulationConfig
```

Now we'll define some physical constants, and a conversion factor from Hertz to electron-volts.

```
mu0, c, eps0, h, hz_eV :: Real
-- | Vacuum permeability
mu0   = 4.0e-7*pi

-- | Speed of light
c     = 299792458.0

-- | Permitivity of free space
eps0  = 1.0/(c*c*mu0)

-- | Planck's constant
h     = 1.054571628e-34

-- | Conversion factor from hertz to electron-volts
hz_eV = 2.4180e14

-- | The imaginary unit
i :: Complex Real               -- LOL
i     = 0.0 :+ 1.0

-- | Default configuration used to generate plot in article
defaultConf :: SimulationConfig
defaultConf = SimulationConfig
              {
                _d_omega = 2.0*pi*hz_eV*0.04
              , _dp      = 3.33564e-30*10.0
              , _dt      = 1.0e-9/(2.0*c)
              , _e0      = 861000000.0
              , _nt      = 1000000
              , _nw      = 200
              , _omega0  = 2.0*pi*hz_eV*2.0
              , _tau     = 20.0e-15
              }
```

Hooray! We're done with the boilerplate. Now we can move onto the meat of the simulation. Our pulse will of course have a sinusoidal component, but our envelope will be give by a $\sin^2$ function.

```
-- | Given a configuration and a timestep, return the value of the electric
-- field E(t) at that step.
pulse :: SimulationConfig -> Int -> Real
pulse conf n = let t = fromIntegral n*conf ^. dt
               in if t <= conf ^. tau
                  then conf^.e0*cos(conf^.omega0*t)*sin(pi*t/conf^.tau)**2
                  else 0.0
```

We'll now define strict versions of `foldl` and `foldl1`. These will cut down on memory usage a lot, and make things faster to boot. Haskell's stream fusion is quite amazing: the maximum memory consumption of this process on my laptop is well under 2 gigabytes.

```
-- | Strict version of foldl
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' _ z []     = z
```

```
foldl' f z (x:xs) = let z' = z `f` x
                    in seq z' $ foldl' f z' xs

-- | Strict version of foldl1
foldl1' :: (a -> a -> a) -> [a] -> a
foldl1' f (x:xs) = foldl' f x xs
foldl1' _ _ = error "foldl1'"
```

Hey, that wasn't so bad. When are things going to start going the way of Figure 1? Now. We'll write the whole simulation as one large function, broken up into a few local ones. We want to know the evolution of the system's net dipole moment over time, so we'll simulate each type of molecule in parallel, and add it all up at the end. By the way, doing this all in parallel is as easy as changing one line of code - all we do is change `dips` to ``dips `using` parList rdeepseq`` in the toplevel expression of `simulateAtoms`. (`dips` is a list of `vectors`, one for each type of molecule's time-evolved dipole moment.)

I originally wrote this code in Fortran, and Haskell's logic and control flow is much tighter than the original, but the numeric parts are a bit more verbose, due to the need to manually convert from `Int` to `Real` and `Real` to `Complex Real`. `omegaW` is a particular molecule's resonant frequency, and the various `omega_n` are time-discretized and interpolated versions of Ω. Since the code is adapted from a larger simulation which used a larger density matrix, mentally replace ρ_{11} with ρ_{gg}, ρ_{12} with ρ_{ge}, and ρ_{22} with ρ_{ee}.

So, here's the meat of the simulation:

```
-- | Simulates a system of non-interacting molecules with different resonance
-- frequencies according to the values in conf.
simulateAtoms :: Complex Real   -- ^ initial rho11
              -> Complex Real   -- ^ initial rho12
              -> Complex Real   -- ^ initial rho22
              -> Real           -- ^ initial dipole moment
              -> SimulationConfig -- ^ simulation configuration
              -> U.Vector Double  -- ^ net dipole moment
simulateAtoms rho11 rho12 rho22 dipole conf =
    -- Calculate net dipole moment
    foldl1' (\d1 d2 -> U.zipWith (+) d1 d2) (dips `using` parList rdeepseq)
    where -- Time-evolved dipole moments of the different types of molecules in the system
          dips = map (\w -> simulateAtom rho11 rho12 rho22 dipole w) [1..conf^.nw]

          -- Perform one step of an RK4 integration
          step rho11 rho12 rho22 dipole w n =

              let -- Unpack the configuration and convert as needed
                  dt' = conf^.dt :+ 0
                  w' = fromIntegral w
                  nw' = fromIntegral $ conf^.nw
                  (omega0',d_omega') = (conf^.omega0, conf^.d_omega)

                  -- Resonance frequency for this atom
                  omegaW = (omega0'-d_omega'+2.0*d_omega'*(w'-1)/(nw'-1)) :+ 0.0

                  -- Pulse value at the present and previous timestep
                  p_n = pulse conf n
                  p_n1 = pulse conf (n-1)
```

```
-- Rabi frequency at time n-1
omega_n = (p_n1*(conf^.dp)/(sqrt 3.0*h)) :+ 0.0

-- First step
r11_1 = rho11
r12_1 = rho12
r22_1 = rho22

k1_11 = -i*dt'*omega_n*(conjugate r12_1-r12_1)
k1_12 = -i*dt'*(omega_n*(r22_1-r11_1)-omegaW*r12_1)
k1_22 = -i*dt'*omega_n*(r12_1-conjugate r12_1)

-- Second step
r11_2 = r11_1 + k1_11/2.0
r12_2 = r12_1 + k1_12/2.0
r22_2 = r22_1 + k1_22/2.0

-- Interpolated Rabi frequency
omega_n05=(p_n1+p_n)*conf^.dp/(2.0*sqrt(3.0)*h) :+ 0.0

k2_11 = -i*dt'*omega_n05*(conjugate r12_2 - r12_2)
k2_12 = -i*dt'*(omega_n05*(r22_2-r11_2)-omegaW*r12_2)
k2_22 = -i*dt'*omega_n05*(r12_2-conjugate r12_2)

-- Third step
r11_3 = r11_1 + k2_11/2.0
r12_3 = r12_1 + k2_12/2.0
r22_3 = r22_1 + k2_22/2.0

k3_11 = -i*dt'*omega_n05*(conjugate r12_3 - r12_3)
k3_12 = -i*dt'*(omega_n05*(r22_3-r11_3)-omegaW*r12_3)
k3_22 = -i*dt'*omega_n05*(r12_3-conjugate r12_3)

-- Rabi frequency at present timestep
omega_n1 = (p_n*conf^.dp/(sqrt 3.0*h)) :+ 0.0

-- Fourth step
r11_4 = r11_1 + k3_11
r12_4 = r12_1 + k3_12
r22_4 = r22_1 + k3_22

k4_11 = -i*dt'*omega_n1*(conjugate r12_4-r12_4)
k4_12 = -i*dt'*(omega_n1*(r22_4-r11_4)-omegaW*r12_4)
k4_22 = -i*dt'*omega_n1*(r12_4-conjugate r12_4)

-- Integrate the relevant components of the density matrix
integrate r0 k1 k2 k3 k4 = r0+(k1+2.0*(k2+k3)+k4)/6.0
rho11' = integrate r11_1 k1_11 k2_11 k3_11 k4_11
rho12' = integrate r12_1 k1_12 k2_12 k3_12 k4_12
rho22' = integrate r22_1 k1_22 k2_22 k3_22 k4_22

-- Integrate dipole moment, weighted according g(\omega)
```

```
                dipole' = dipole+realPart rho12'*(sin(w'*pi/nw'))**2
            in (rho11', rho12', rho22', dipole', n+1)

          -- Simulate one atom for n timesteps
          simulateAtom r11 r12 r22 dpl w = nextDipole
              where step' (rho11', rho12', rho22', dipole', n') = step rho11' rho12' rho22' dipole' w n'
                    nextDipole = U.map (^._4) $ U.iterateN (conf^.nt) step' (r11, r12, r22, dpl, 0)
```

It's unfortunate that there's so much code in the step function, but there's not a lot you can do when you need to use a lot of fairly ugly formulas. We just need a `main` to make this all go. We'll take the output file as an argument.

```
main :: IO ()
main = do [out] <- getArgs
          let dipole' = simulateAtoms (1.0 :+ 0.0) (0.0 :+ 0.0) (0.0 :+ 0.0) 0.0 defaultConf
              dt' = defaultConf^.dt
          handle <- openFile out WriteMode
          hSetBuffering handle (BlockBuffering Nothing)
          let writeLine n = TIO.hPutStrLn handle
                          $ T.concat
                          $ map T.pack
                                [show $ 1e15*dt'*fromIntegral n, " ", show $ dipole' U.! n]
          mapM_ writeLine [0..defaultConf^.nt-1 :: Int]
          hFlush handle
          hClose handle
```

4 Results

I compiled and ran the resulting program as follows on my Core i7 laptop:

```
$ ghc -O2 -fllvm -threaded -o atoms-hs Atoms.hs
$ time ./atoms-hs out.dat +RTS -N8

real  0m53.554s
user  2m40.630s
sys 0m24.393s
```

I plotted the output with xmgrace, generating the following plot.

Everything is working as expected.

4.0.1 Thoughts on Using Haskell

Overall, using Haskell was an enjoyable experience. I made a few typos when translating from the original Fortran, but it was easy enough to figure out what was wrong with `Debug.Trace`. It was quite easy to parallelize as well. A minor pain point was `^.` from lens, which doesn't work as consistently the `%` notation does in Fortran due to other operators with the same precedence. I had no problems with excessive memory usage or garbage collection.

The only two significant complaints I have are the amount of numeric conversions and the slow IO (which I'm sure is due to my inexperience with non-trivial IO in Haskell). Most of the runtime of this simulation is spent writing the `Vector` of a million timestepped dipole values to an ASCII text file. I can't share the original code, but the actual calculation in Haskell is only 5-10% slower than the calculation using Fortran. The IO is where Haskell loses ground.

Once I figure out how to do faster IO, I'd be very interested in doing some more experiments with fancier numerical gizmos, like multigrid, spectral, or finite element methods. The more sophisticated

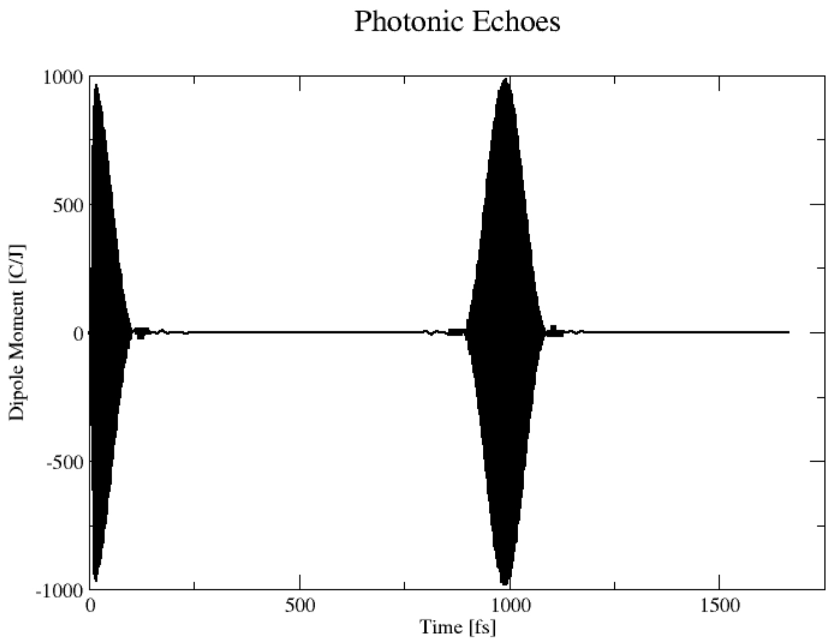


Figure 4: Results of running the above code.

data structures in those methods would make writing such things in Fortran pure misery, and even C++ (which is a "too high-level" to many in this line of work) would be painful.