`Quantum::Entanglement`
Perl gets (quantum) physical

Alex Gough

June 28, 2001

Abstract

Quantum mechanics is one of the stranger things to have emerged from physics over the last hundred years. It has led the way to new understanding of a diverse range of fundamental physical phenomena and, should recent developments prove fruitful, could also lead to an entirely new mode of computation where previously intractable problems find themselves open to easy solution.

The `Quantum::Entanglement` module attempts to "port" some of the functionality of the universe into Perl. Variables can be prepared in a superposition of states, where they take many values at once, and when observed during the course of a program, will collapse to have a single value. If variables interact then their fates are linked so that when one is observed and forced to collapse the others will also collapse at the moment of observation.

The module also makes possible simple complex logic gates, such as $\sqrt{\neg}$, and allows for some algorithms from quantum computing to be run, for instance Shor's method for factoring numbers. As well as sensible programs the module also allows you to play hangman against an opponent who cannot know the word you are trying to guess, provide solvable Countdown maths puzzles and run subroutines in multiple universes.

1 Quantum Mechanics in a Nutshell

1.1 The Ghost in the Atom

At the start of the last century, physics was in trouble. The tattered edges of the theories which had served science so well for so long were becoming exposed by increasingly accurate experiments. The mainstays of thermodynamics, electromagnetism and Newtonian mechanics could not explain, or more importantly predict, the observed spectra of atoms and hot bodies.

Ever alert for the call to action, physicists soon assembled a new version of mechanics, resting on a small set of simple assumptions and equations which was to revolutionise our understanding of the universe. From this theory sprang startling results: physical quantities such as energy were now able to take on only a set of precisely defined values; particles like electrons were also waves but waves like light were also particles, systems could exist in a ghostly overlay of many different states at once, but only if you didn't look at them and cats the world over feared for their existential well-being.

Hyperbole aside quantum theory is not quite as complicated as it sometimes seems. At its root is a simple equation and a few rules which must be applied when calculating physical outcomes. The rest of this section presents a brief overview of this structure, do not worry if you cannot follow, just skip ahead to the perl.

1.2 Solving an Equation

You can safely skip this section...

Classically, a particle can be thought of as being a blob of stuff, with a precisely defined set of values associated with its position, direction of motion, speed, energy and so on. In quantum mechanics, this version of events is overturned. A particle instead becomes a wave, or curve, which satisfies an equation, its position and energy must be extracted from this equation via a bit of mathematical trickery which is outlined below.

Because physics is mostly concerned with the energy a particle can have, we usually start by finding sets of functions which satisfy Schrödinger's famous equation,

$$[-\hbar^2 \frac{\partial^2}{\partial x^2} + \mathcal{V}]\phi_n(x) = E_n \phi_n(x) \tag{1}$$

where E_n must be a real number, corresponding to the energy of a particle in the state $\phi_n(x)$. We call E_n an eigenvalue, and $\phi_n(x)$ an eigenfunction of the equation. Generally $\phi_n(x)$ will be something like $\sin(nx/a)$ or $\cos(nx/a)$, it is also this equation which forces E_n to take quantised values which depend on the particular situation we are looking at. We then ask what $\phi_n(x)$ actually *represents*. We find that if we square this function, then if we know that a particle has an energy of E_i then if we look at where the particle is, we will see it at a position $x = k$ with a probability of $|\phi_i(k)|^2$.

Sometimes things get a little bit more complicated though. What if we want to know what the momentum of the particle is, rather than its energy? Instead of finding solutions to equation 1 we need to find solutions to another equation

$$-i\hbar \frac{\partial}{\partial x} \phi'_m(x) = P_m \phi'_m(x) \tag{2}$$

which could be different from the solutions we found to equation 1. But what if we want to know both? In general it turns out that we cannot, but we can measure one (the momentum say) for which we get a value P_k, which tells us that the particle must have a probability distribution described by $|\phi'_k(x)|^2$. When we then measure the energy of the particle, we cannot be sure of the value we will get. We must add together possible solutions to equation 1 (using lots of different values of n) so that the curve we get is the same as $\phi'_k(x)$, or

$$\phi'_k(x) = a_0\phi_0(x) + a_1\phi_1(x) + a_2\phi_2(x) + \ldots. \tag{3}$$

In words, we form a superposition of possible $\phi_n(x)$. When we then measure the energy of our particle, we get E_0 or E_1 and so on, each with a probability given by a_0^2 or a_1^2 (so long as $\sum_{i=0}^{\infty} a_i^2 = 1$). Of course, once we've measured the energy, we again know exactly which energy state the particle is in, although we will no longer be able to assign a unique value to the momentum of the particle.

Whoa, you might want to read this last bit.

When in a superposition like equation 3 we cannot treat the particle as if it were in a particular state, but instead must think of it as occupying every possible state in the equation at the same time. Only when we observe some property of the particle does it take on a single, well defined state.

Or, put more briefly, sometimes a quantity can have many possible values at once, the chance of it collapsing to have a one particular value is given by the square of a coeficient in a superposition equation. Even more briefly: things are strange.

2 Creating Superposed States

It is quite hard to provide a complete version of the above in perl, so we need to make some compromises. Instead of solving thousands of equations each time we want to do something, we will assume that only one set of values (or solutions) exist which will be provided by the user, we will also forget entirely about eigenfunctions (the $\phi_n(x)$ above). This still allows us to give variables multiple values at the same time, and to control which values they collapse to when observed by using probability amplitudes.

This means that we can create a variable in the following superposition of states

$$\texttt{\$var} = a_0|0\rangle + a_1|1\rangle + a_2|2\rangle, \tag{4}$$

(where the $|\ldots\rangle$ pattern indicates a possible value of the variable) we can then use `$var` throughout a program as we would any normal variable, but so long as we do not try to find out the actual value of `$var` every possible value it can take will be present in every calculation, this also means that the results of any calculations involving the variable will have many possible values.

2.1 The `entangle()` function

The `Quantum::Entanglement` module adds an `entangle()` function to perl, this takes a list of amplitudes and values and returns a variable in a superposition of values, saying

```
$var = entangle( $a0 => 0,
                 $a1 => 1,
                 $a2 => 2 );
```

creates the superposition of values in 4 above.

2.2 Observation and Collapse in Perl

We then need to decide what happens when we observe our variable, and what exactly we mean by *observe*? Taking a broad definition of observation as being "anything that reveals the values which a variable has" seems about right. Perl provides us with many ways of doing this, there are the obvious acts of printing out a variable or testing it for truth, but also operators such as `eq` or `>=` tell us something.

How do we decide which values a variable collapses to when observed? Well, each possible value has an associated probability amplitude, so all we need to do is build up a list of possible outcomes, add up the amplitudes, square the result, then use this to bias the value (or values) that the variable collapses to.

For the superposition in 4, if we were to say

```
if ($var == 1) {
  print 'equal to 1';
}
else {
  print 'not equal to 1';
}
```

we would see '`equal to 1`' printed with a probability of $a_1^2/(a_1^2 + (a_0 + a_2)^2)$ and '`not equal to 1`' with a probability of $(a_0 + a_2)^2/(a_1^2 + (a_0 + a_2)^2)$.

Now an interesting possibility arises, what if $a_0 = -a_2$, can we do this? The answer is "yes", and this means that the collapse of our variables in different circumstances will be different, if we had just wanted to `print` our variables, then there is a chance that either value will be output, but if we test our superposition for "oneness" we will find that it is *always* the case.

2.3 Hangman

We can now start to write interesting programs, even before we hit the really weird stuff. You have probably heard of the game "twenty questions", where one player thinks of an object, and must answer 'yes' or 'no' to twenty questions from the other before he has to guess what the object is. If you are careful, it is possible to play this without actually thinking of an object, answering the questions any way you want so long as you can eventually think of an object which is consistant with your answers at the end of the game (some people might call this cheating though). The following program allows you to play hangman against an opponent who does not know which word you are trying to guess, until that is, you have made some guesses.

First of all, we read in a list of words of a given length (to make the game a little bit easier)

```
#!/usr/bin/perl -w
use Quantum::Entanglement;
my @words;
```

```
print "Reading word list...";
open DICT, '</usr/share/dict/words' or die "dict: $!";
while (<DICT>) {
  chomp;
  push(@words,1,lc($_)) if length($_) == $ARGV[0];
}
close DICT; print "done\n";
```

you may notice that the array `@words` has two elements added for each word, the first is the probability amplitude we want to assign the word, and the second is the word itself, this allows us to create a single word which is a superposition of all possible words by saying

```
my $word = entangle(@words);
```

we then ask our player to guess a letter (or range of letters with `[a-e]`) as well as providing a way to give up,

```
while (<STDIN>) {
  chomp;
  print "So, you give up... $word\n" and last if /tell/;
```

If the player gives up when there are still many possible values of the word remaining, printing out `$word` will cause a collapse to a single word. If they want to continue playing, we test our word with a regular expression[1], this will select two groups of possible results, those words which match our guess, and those that do not, and will then select one of them at random causing `$word` to collapse so that it can *only* be those which match (or not, depending on the direction of collapse), this all happens transparently.

```
  if (p_op($word, '=~', qr/$_/)) {
    print " \$word does contain $_\n";
  }
  else {
    print " \$word does not contain $_\n";
  }
}
```

3 Entanglement and non-local Collapse

Another result to emerge from the equations of quantum theory was that when two quantum particles (such as photons) interacted, their wavefunctions (the $\phi_n(x)$) became linked with a single wavefunction describing the entire system ($\phi_{n,m}(x_1, x_2)$). This connection meant that when one of the particles was measured and collapsed to some definite state, the other would also collapse, even if

[1] As regex binding cannot be overloaded, we need to cheat a bit.

separated by great distances. This manifests itself as a greater degree of correlation between entangled particles than would be expected if the two particles collapsed at different times (the Bell inequality) or did not become entangled when they interacted.

3.1 Entanglement in Perl

Whenever superposed variables interact, or are involved in calculations, the results of these as well as the variables themselves become entangled. This means that they will all collapse at the same time, so as to remain consistant with their history. For instance, the following

```
$var = entangle( 1 => 0, 1 => 1 );
$result = $var * 2;
if ($result == 2) {
  print "\$var equals 1\n" if $var == 1;
} else {
  print "\$var equals 0\n" if $var == 0;
}
```

will cause **$var** to always collapse to 1 if **$result** was 2, or 0 if **$result** was not equal to 2. Although we cannot predict which way **$result** will collapse, once it has, we can predict what **$var** will be because of entanglement.

4 Simple Complex Logic

If entanglement isn't enough to scare you, quantum theory also makes possible new logical operations, first though, we need to expand the what we can do with a superposition.

4.1 Complex Amplitudes

If we can have negative values as the coefficients of our superpositions, it seems sensible that we can also use complex numbers. The only difference being that instead of just squaring the number when working out our probability, we need to square the *size* of the number. (eg. $|1 + 2i|^2 = 5 = |1 - 2i|^2$.)

4.2 Generic Logic Gate

Figure 1 shows a generic logic gate which takes a single input, and transforms it into a single output. The coefficients a_{ij} give the probability that a given input will be transformed into a given output. If $a_{00} = a_{11} = 1$ and $a_{01} = a_{10} = 0$ then the gate does nothing (a pass-though), if $a_{01} = a_{10} = 1$ and $a_{00} = a_{11} = 0$ then the gate will invert its input (a NOT gate). We can also allow $a_{ij} = 0.5$, which will act as a randomising device producing an output stream which is not correlated with its input. If we chain two of these together, so that the input of

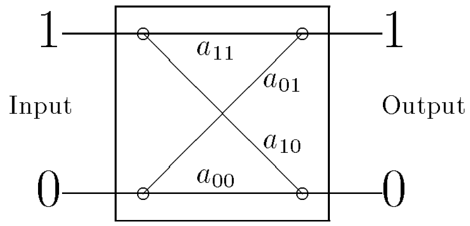

Figure 1: A generic logic gate, capable of accepting *one* input, which can take either of two possible values. The coefficients a_{ij} indicate the probability that a given input state will be transformed into a given output state.

one becomes the output of the second, then we continue to get perfectly random noise.

4.3 Complex Logic Gate

Things become interesting if instead of using real numbers and definite inputs (a classical picture), we allow a_{ij} to be complex numbers, let the input and output be a superposition of input states and modify the workings of our gate so that the probabilities (a_{ij}) multiply the coefficients of our superposition equation, so that an input state of

$$b_0|0\rangle + b_1|1\rangle$$

becomes an output state of

$$(a_{00}b_0 + a_{10}b_1)|0\rangle + (a_{11}b_1 + a_{01}b_0)|1\rangle.$$

Now, if we let $a_{00} = a_{11} = i/\sqrt{2}$ and $a_{01} = a_{10} = 1/\sqrt{2}$, an input state of $|0\rangle$ will be transformed into a superposition of two output states

$$\frac{i}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

which if observed, will collapse with equal likelyhood to either 0 or 1 (as the size of the coefficients for each state is the same). If we use only one of these gates, then we see the same random noise as we did with the classical random gate above.

What if we restrain ourselves and rather than looking at the output of the first gate, send the superposed state though an identical gate? We find that the output of the second gate will be in the state

$$\left(\frac{i}{\sqrt{2}}\frac{i}{\sqrt{2}} + \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\right)|0\rangle + \left(\frac{i}{\sqrt{2}}\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}\frac{i}{\sqrt{2}}\right)|1\rangle$$

which simplifies to

$$\left(-\frac{1}{2} + \frac{1}{2}\right)|0\rangle + \left(\frac{i}{2} + \frac{i}{2}\right)|1\rangle = 0|0\rangle + i|1\rangle$$

if the output of the second gate is observed, then as only one state has a non-zero coefficient, it *must* collapse to 1. Whereas the classical version of this gate had the same effect when two were chained together, the quantum version produces noise if one is used, or negates its input if two are chained together, in other words, we have a $\sqrt{\neg}$ (square root of NOT) gate, a function entirely absent from classical computing.

$$\sqrt{\neg}(0) = |noise\rangle$$

$$\sqrt{\neg}(|noise\rangle) = 1$$

4.4 $\sqrt{\neg}$ in Perl

The `Quantum::Entanglement` module allows subroutines to create new states based on the current set of states by using the function `q_logic` and a subroutine. The following program implements a $\sqrt{\neg}$ gate:

```
#!/usr/bin/perl -w
use Quantum::Entanglement qw(:DEFAULT :complex);

my $foo = entangle(1 => 0);
$foo = q_logic(\&root_not, $foo);
$foo = q_logic(\&root_not, $foo);

print "\$foo is true!\n" if $foo;

sub root_not {
  my ($prob, $val) = @_;
  return( $prob * i / sqrt(2) , $val,
      $prob / sqrt(2) , !$val ? 1 : 0);
}
```

The output is always `$foo is true!`.

4.5 $\sqrt{\neg}$ in the Real World?

The $\sqrt{\neg}$ gate is more than a mathematical sleight of hand as devices exist which perform this function. The semi-silvered (or two-way) mirrors, much beloved of gangsters and politicians for their car windows, can be designed so that a photon has a fifty-fifty chance of passing through the mirror or being reflected. Quantum physically, this means that a photon follows a superposition of *both* possible paths (until observed). When the light leaving the mirror is carefully bent back on itself (see figure 2) and sent through another two-way mirror the two superposed paths are recombined causing the photon to emerge from the second mirror in one direction only. If the path taken by the photon is determined (even when inferred by its absence from another path) it can emerge from the final mirror in *either* direction. This can only be explained if it is

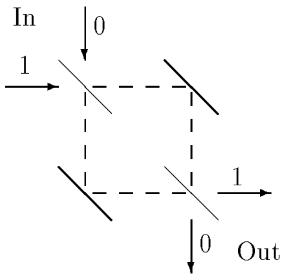

Figure 2: Two semi-silvered mirrors acting as $\sqrt{\neg}$ gates, a single photon passing through the system takes both paths (dotted lines) simultaneously

assumed that the photon follows both paths through the equipment, and only if the route it takes is not examined.

5 More Useful Algorithms

The fundamental links between quantum and classical computing give us new insights into what is and isn't computable (the Turing machine functions are a subset of the functions of the universal quantum computer) which is certainly interesting physics and comforting for computer scientists, but is quantum computing actually useful for anything? About ten years after the subject had its genesis, algorithms were discovered which could outperform their classical counterparts. This translates well into research funding so expect to hear more in the future.

5.1 Factoring Numbers

Factoring large numbers is well known as a hard problem, it is also an area which is begging for a more efficient algorithm than classical brute-force methods can manage as many widely used encryption schemes gain their security from the difficulty of discovering two large prime numbers given only their product.

5.1.1 Shor's Algorithm

Back in 1995, Peter Shor discovered an algorithm which was able to factor large numbers. By abusing the periodic nature of certain mathematical functions, coupled with Fourier transforms and the parallelism which can be obtained from superposed states, it is possible to factor numbers in nearly constant time.

5.1.2 Shor's Algorithm in Perl

The `Quantum::Entanglement` module can run Shor's algorithm. An example of this, along with a short explanation, is included in the distribution of the module from the CPAN. It does not quite achieve the desired speed improvement but this is the price of simulation. The following is an attempt to factor 21 using a PII-400 and the zsh `time` builtin:

```
lintilla% time ./shor.pl 21
Performing initial classical steps:
Using q:512, x:4
Starting quantum steps
Performing F = x**|a> mod n
$register2 collapsed to 1
Finding period of F (this is where you wish for a QCD)
Period of F = x**|a> mod n is 2
3 * 5 and 21 might share a gcd (classical step)
3, 1 could be factors of 21
./shor.pl 21  125.67s user 0.04s system 89% cpu 2:20.14 total
```

6 Breaking the Laws of Physics

6.1 Controlling Collapse

Given that we have a framework which closely mirrors some of the laws of physics, there seems to be no good reason why we might not sometimes want to override these rules. We may occasionally want to force a superposition to collapse to a given value, perhaps when preparing it for use in a more complex problem, we might also want to cheat at hangman.

The module provides a flag `$::conform` which, when set to a true value, will cause any operation which could return either true or false values, to always return truth, if possible, and retain only those states which cause the true value. Variables which are entangled with the one which collapsed also collapse as before. A method of saving the state of a superposition is also available so that it can be restored after a collapse. This can be used on multiple variables so that the entangled relationships between them are retained.

6.2 Playing Perfect Hangman

If the program listed in section 2.3 is modified so that it contains

```
 $Quantum::Entanglement::conform = 1;
```

near the start, this will cause the collapse of `$word` to always follow the guesses of the player, if at all possible. On the other hand, by changing the test we use on `$word` to `'!~'` then we find ourselves playing against a dealer who tries his best to make our guesses fail.

7 Internal Details

> "Imagine a box, a really big, really black box, in a black room, with no lights, and maybe no box."

Writing this behavior into Perl presents an interesting challenge, a means of representing a superposition is required, as is some way of allowing different variables to know about each other, without creating circular references which would stand in the way of garbage collection. We also need a means to cause collapse, as well as a robust mechanism for dealing with both real and complex numbers. Thankfully Perl provides a rich set of ingredients which can more than satisfy these requirements without making the job so hard that it becomes impossible.

7.1 Overloading

Perl has a fairly good overloading mechanism, it allows the working of most (although not all) operators, and some core functions, to be redefined for objects `bless`ed into a given package. Each entangled variable is simply an object, for which all operators have been modified so as to act in parallel on the values it can take. Collapse is caused by any operator which identifies some properties of the superposition, `==`, or stringification say. Operators which do not reveal anything about the state of a variable return a new entangled variable.

7.2 Representation

If we deal with the simple case of two variables, which are entangled with one another after some sort of interaction, we find that for each possible value that the first (a_1 say) can take, the second can take each of its values (b_j), the same being true for a_2 and so on. This can be displayed as shown in figure 3. This is nothing more than a two dimensional table, so is natural to use a list of lists. We have left out the probability amplitudes, but these can just be slipped in the column next to the values with which they are associated.

7.3 Interactions and Entanglement

When two entangled variables interact (usually producing an entangled result), we need to join together both those variables, and all the variables which have interacted with those variables earlier in the program. This is acheived by creating a new table of states, which is the 'cross' of the two previous sets of states. This is illustrated in figure 3.

7.4 Collapse

Using this tabular approach it becomes easy to remove unwanted states when variables are observed so that all variables are modified at the same time. If we

`$a = entangle(...)`	`$b = entangle(...)`	`$c = $a * $b`		
		a_0b_0	a_0	b_0
a_0	b_0	a_0b_1	a_0	b_1
a_1	b_1	a_1b_0	a_1	b_0
		a_1b_1	a_1	b_1

Figure 3: Before any interaction, the possible values of a and b are uncorrelated, afterwards, each value of a must be paired with every possible value of b, values of c then depend on these pairs

`$b == b1`		
$\cancel{a_0b_0}$	$\cancel{a_0}$	$\cancel{b_0}$
a_0b_1	a_0	b_1
$\cancel{a_1b_0}$	$\cancel{a_1}$	$\cancel{b_0}$
a_1b_1	a_1	b_1

Figure 4: Unwanted states are easy to remove, as we must only delete rows from a table, automagically affecting all variables.

imagine that we tested `$b` above, and found that it was equal to b_1, we simply remove all the rows in which it isn't equal to b_1 as shown in figure 4.

7.5 Indirection

Having built up this table, how do we know which data belongs to which variable, and how do we ensure that we can delete unwanted data? This problem is solved by using two levels of references, so that each variable contains a reference to a reference to a list of lists, each variable also needs to know the position its data has in this table, but also needs to know the positions of every other variable, so that when two of these tables are joined together, each variable has its position updated at the same time. This is acheived through each variable also holding a reference to a list of references to these offsets.

By using two levels of referencing, it becomes easy to redirect the target of the middle reference, as happens when two entangled variables interact, requiring that they share the same table of outcomes. When a new variable is created, it has this structure

```
my $var = [\$universe,1,\$offsets];
$offsets->[0] = \ $var->[1];
```

where `$universe` is the same for all variables with which it has interacted. This means that changing the target of `${$var->[0]}` to a new universe, will change the target for all variables inhabiting the old universe. The same holds for the array `@$offsets`. This structure also avoids any circular referencing, which means that variables are destroyed sensibly as nothing outside the program using the module stops variables from going out of scope.

7.6 Limitations in Perl

While Perl's referencing, overloading and easy creation and extension of complicated data structures makes modules like this easy to write, it still lacks features which could make the job even easier.

It is not currently possible to define new operators and some key operations (such as `=~`) cannot be overloaded. A few of the automatically generated overloaded operators cannot be distinguished from one another within the module doing the overloading, which makes it hard to producing warnings consistantly, and can lead to ugly code elsewhere.

Perl lacks 'friendly' multi-dimensional data structures, making it hard to take slices along two different dimensions of a matrix without writing excess code to control doing so. It is also not possible to easily define array slices with constant steps (`2,4,6,8...`). On balance though these are small complaints and mostly the price of the very general approach of Perl.

8 Into the Future ...

The following section details some potential improvements which might find their way into the module in the future. Each tries to overcome current limitations or extend the range of phenomena simulated by the module.

8.1 Delaying Collapse

As it stands the observer which causes the collapse of variables in any program is the program itself. This misses the fun which could be had by making the person running the program the observer, effectively keeping everything inside the program in a superposition of states until some output is produced. How to deal with situations like

```
$bar = entangle(1=>0, 1=>1);
if ($bar == 0) {   # no output produced
  $foo = $bar * 2;
}
else {             # output produced
  print "I think \$bar != 0\n";
  $foo = $bar + 1;
}
print "$foo\n";
```

poses a little problem. Do we run the entire program, capturing all potential output, then at the end collapse and dump one of the possible versions or do we assume that any conditional branching which causes different output to be produced always acts as instant observation?

8.2 Parallel Execution

The point above draws us towards parallel execution. It would be nice to play about with a source filter, and some magic, to allow all possible execution paths to be followed. Perhaps allowing them to interfere with each other, as happens with the real $\sqrt{\neg}$ gate in figure 2. This could also allow for execution to be spread over multiple processors or machines, although Perl is not well suited to HPC applications.

8.3 Uncertainty

Recalling the discussion in section 1.2, in the real world (which is weirder than we can be) the possible values which a measurement can give depend on what else has been measured about a particle. For instance, we saw that we cannot know both the energy (E_n) and momentum (P_i) of a particle. This known as the *uncertainty principle* and it would be lovely if it could be added to Perl.

The fully fledged functional version is a little beyond Perl, but by using matrix operators, each of which have different sets of eigenvalues, it should be possible.

8.4 It's Physics Jim, but not as we know it

This module does fall short of physical reality in a few important areas, some of which are listed below:

- **No eigenfunction-like behaviour** All operators share the same set of eigenfunctions, in real QM this is sometimes not the case, so observing one thing would cause some other thing (even if already observed) to fall into a superposition of states again.
- **Certain observables cannot simultaneously have precisely defined values** This follows from the point above. The famous uncertainty principle arises as position and momentum have different sets of eigenfunctions. In this module, it is always possible to collapse the system so that a value is known for every entangled variable.
- **Perl is not a quantum computing device** Perl, alas, is currently only implemented on classical computers, this has the disadvantage that any quantum algorithm will run in exponential time. This might be remedied in future releases of Perl. Just not anytime soon.
- **Quantum information cannot be copied with perfect fidelity** It is impossible to perfectly clone a real entangled state without 'damaging' in some way either the original or the copy. In this module, it is possible as we have special access to the states of our variables.
- **Cannot generate perfectly random numbers** It is well known that classical computers cannot produce a perfectly random sequence of numbers, as this module runs on one of these, it also suffers the same fate. It

is possible to give a classical computer access to a perfect random number generator though (essentially by linking it to a suitable physical system) in which case this is no longer a problem.

9 Just another thing

```
#!/usr/bin/perl
use Quantum::Entanglement qw(:DEFAULT :complex);
$language = entangle(1,'python',1/i,'C',i*i,'perl',
                     1/i**2,'Java',i**5,'C#');
print "Just another $language hacker,\n" if $language eq 'perl';
```

10 More Information

The 1985 paper "Quantum theory, the Church-Turing principle and the universal quantum computer" [2] by David Deutsch which 'launched' quantum computing is available from the Oxford Center for Quantum Computation, the rest of their site has good number of introductory articles. Another paper worth reading is "Machines, Logic and Quantum Physics"[3] by David Deutsch, Artur Ekert and Rossella Lupacchini.

The module distribution (available from the CPAN) comes with a few short demonstrations of what the module can do, along with plenty of explanation (including Shor's algorithm from section 5.1.2). If you do anything interesting with the module, or have any questions, do feel free to contact me: `alex@rcon.org`.

[2] `http://www.qubit.org/resource/deutsch85.ps`
[3] `http://xxx.lanl.gov/abs/math.HO/9911150`