Appendix E

Reversible Schrödinger wave simulation

This appendix gives the complete code for SCH, the reversible simulator of the Schrödinger wave equation that was mentioned in §8.5.6 (p. 221), and in §C.5 (p. 289). We give the C, R, and PISA versions of the program.

E.1 Derivation of discrete update rule

Here we derive a naive approximate method for simulating Schrödinger's equation in discrete, reversible fashion. Later we will derive some alternatives.

A bit of history. Most of the following is original work, except that the final key idea, for making the simulation exactly reversible, is something that I learned about from Margolus in personal discussions. This trick apparently originated with Fredkin and Barton in 1975, in the context of their own work (in collaboration with Richard Feynman) on a discrete reversible Schrödinger equation update rule. (I was not aware of this work when I reinvented part of it in the early versions of my own simulation.) The story of the serendipitous discovery of this trick is told in Fredkin 1999 [63], which also describes their version of the discrete rule in some detail.

Now, let us begin our derivation. We start with the full general form of the wave equation when expressed in a state space in which the eigenstates correspond to particle positions. It is possible to describe a system's state space in many other ways, but this way seems most straightforward and natural to those not immersed in quantum physics. It also lends itself to visualization of the wave function on a graphics display.

Here is the Schrödinger equation in its full glory:

$$-\frac{\hbar^2}{2}\sum_{j=0}^{N-1}\frac{1}{m_j}\frac{\partial^2}{\partial x_j^2}\Psi(\vec{x},t)+\mathcal{V}(\vec{x},t)\Psi(\vec{x},t)=i\hbar\frac{\partial}{\partial t}\Psi(\vec{x},t).$$

It requires some explanation for the general reader. $\hbar$ is Planck's constant over 2π, $1.055\times 10^{-34}\mathrm{J\cdot s}$. N is the number of positional degrees of freedom, $3n$ for n particles in 3 dimensions. m_j is the mass associated with the jth degree of freedom, *e.g.* in 3 dimensions the mass of particle $\lfloor j/3\rfloor$. $\vec{x}$ is a vector of all particle position coordinates, and x_j is particular position coordinate. t is time. $\mathcal{V}$ is a potential energy function which is a function of the positions of all particles and optionally (if representing a time-dependent potential) the time. Ψ is the wave function itself, a function of the positions of all particles and of time; its value is generally a complex number with both real and imaginary parts, $\Psi=\Re\Psi+i\Im\Psi$. The imaginary unit $i=\sqrt{-1}$ appears on the right-hand side of the equation. The magnitude of $\Psi(\vec{x},t)$, that is, $\Re^2\Psi+\Im^2\Psi$, is, when normalized, thought of as the probability density of finding the system in state $\vec{x}$ at time t.

The important point to note, from the perspective of a dynamical simulation, is that the equation describes how the wave function evolves over time, via the partial derivative with respect to time that appears on the right side of the equation.

Now that we've seen the full form of the equation, let's proceed to strip it down to a slightly simpler form. First, we'll get rid of the summation over the $\vec{x}$ vector; since we are restricting ourselves for the moment to one particle in one dimension, there is only one coordinate x, and only one mass m.

$$-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\Psi(x,t)+\mathcal{V}(x,t)\Psi(x,t)=i\hbar\frac{\partial}{\partial t}\Psi(x,t).$$

Next, rather than writing the (x,t) arguments to Ψ and $\mathcal{V}$ repeatedly, let them be understood.

$$-\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2}+\mathcal{V}\Psi=i\hbar\frac{\partial\Psi}{\partial t}.$$

Next, we rewrite partial derivative operators $\partial/\partial v$ as d_v/dv where d_v is a differential operator meaning "the infinitesimal change in the given quantity when variable v changes by an infinitesimal amount dv and everything else is held constant".

$$-\frac{\hbar^2}{2m}\frac{d_x^2\Psi}{dx^2}+\mathcal{V}\Psi=i\hbar\frac{d_t\Psi}{dt}.$$

This notational change will allow us to extricate the differentials in the numerator and denominator of a partial derivative operator from each other.

Now, we solve for $d_t\Psi$, the t subscript reminding us that this is the $d\Psi$ that is associated with dt in the original expression $\partial\Psi/\partial t$.

$$d_t\Psi = \frac{dt}{i\hbar}\left(-\frac{\hbar^2}{2m}\frac{d_x^2\Psi}{dx^2} + \mathcal{V}\Psi\right). \tag{E.1}$$

This equation gives us the infinitesimal change in $\Psi(x,t)$ at a point x as time increases by an infinitesimal amount dt and the point x is held constant. This is the most direct statement of how Ψ evolves over time.

Now we come to our first approximation. So far we have treated time and space as continuous: dt means an infinitesimally small amount of time, and dx an infinitesimally small distance in space. However, in our simulation we will *discretize* time and space into points with a minimum, non-infinitesimal distance between them, so that we only have to represent the state of the wave function at a finite number of points, and so that when we advance to the "next" time, there will be a non-infinitesimal change in the wave function state. We indicate this conceptual change by replacing d with Δ, Δx meaning the distance between our discrete points, Δt the distance between our discrete times, and $\Delta_t\Psi$ the change in $\Psi(x)$ in a time Δt.

$$\Delta_t\Psi \approx \frac{\Delta t}{i\hbar}\left(-\frac{\hbar^2}{2m}\frac{\Delta_x^2\Psi}{\Delta x^2} + \mathcal{V}\Psi\right). \tag{E.2}$$

The earlier equation with the differentials (equation E.1) expresses the fact that the two sides of this equation with the deltas approach equality in the limit, as Δx and Δt approach zero. But hereafter in our derivation, we will treat the two sides as being equal even though the deltas are not zero; that is our approximation.

We believe we can prove that this approximation is correct to second order, i.e., that as long as the real rate of change of Ψ is small during the entire interval Δt, and if the minimum wavelength in the represented wave is significantly longer than Δx, then the difference between our computed $\Delta\Psi$ and the actual $\Delta\Psi$ will improve quadratically, proportionately to the square of $\Delta t/\Delta x^2$, as this ratio is decreased.

Now, before we can make further progress, we need to decide how to evaluate the expression $\Delta_x^2\Psi$ at particular points x. In general, by $\Delta_v q$ we mean the change in quantity q, a function of v, when v changes by Δv. However, how are we to define $\Delta_v q(v_k)$ for a particular point v_k? One symmetrical answer is that it is simply $q(v_k + \Delta v/2) - q(v_k - \Delta v/2)$.

Applying this to our particular case, we find (omitting still the ever-present t

argument):

$$\begin{aligned}
\Delta_x^2\Psi(x) &= \Delta_x\Psi(x+\frac{\Delta x}{2}) - \Delta_x\Psi(x-\frac{\Delta x}{2}) \\
&= (\Psi(x+\Delta x)-\Psi(x)) - (\Psi(x)-\Psi(x-\Delta x)) \\
&= \Psi(x+\Delta x)+\Psi(x-\Delta x)-2\Psi(x) \\
&= 2\left(\frac{\Psi(x+\Delta x)+\Psi(x-\Delta x)}{2}-\Psi(x)\right).
\end{aligned}$$

And plugging this into the equation E.2 (considered as an equality) we get:

$$\begin{aligned}
&\Delta_t\Psi(x) = \\
&\quad \frac{\Delta t}{i\hbar}\left(\frac{-\hbar^2}{\Delta x^2 m}\left(\frac{\Psi(x+\Delta x)+\Psi(x-\Delta x)}{2}-\Psi(x)\right)+\mathcal{V}(x)\Psi(x)\right).
\end{aligned}$$

However, rather than sprinkling things like $x+\Delta x$ throughout our equations from here on, let us take it as given that Ψ is always evaluated at points that are separated by integer multiples of Δx, and so we can treat Ψ as a vector with elements indexed by integers k. So hereafter we will replace $\Psi(x)$ by Ψ_k, $\Psi(x+\Delta x)$ by Ψ_{k+1}, *etc.*, and similarly for $\mathcal{V}(x)$ as well.

$$\Delta_t\Psi_k = \frac{\Delta t}{i\hbar}\left(-\frac{\hbar^2}{\Delta x^2 m}\left(\frac{\Psi_{k+1}+\Psi_{k-1}}{2}-\Psi_k\right)+\mathcal{V}_k\Psi_k\right).$$

Now, let's play around with this expression algebraically, and collect together the terms involving Ψ_k:

$$\begin{aligned}
\Delta_t\Psi_k &= -\frac{\Delta t\hbar^2}{i\hbar\Delta x^2 m}\left(\frac{\Psi_{k+1}+\Psi_{k-1}}{2}-\Psi_k\right)+\frac{\Delta t\mathcal{V}_k}{i\hbar}\Psi_k \\
&= -\frac{\Delta t\hbar}{i\Delta x^2 m}\left(\frac{\Psi_{k+1}+\Psi_{k-1}}{2}\right)+\frac{\Delta t\hbar}{i\Delta x^2 m}\Psi_k+\frac{\Delta t\mathcal{V}_k}{i\hbar}\Psi_k \\
&= -\frac{\Delta t\hbar}{2i\Delta x^2 m}(\Psi_{k+1}+\Psi_{k-1})+\frac{\Delta t}{i}\left(\frac{\hbar}{\Delta x^2 m}+\frac{\mathcal{V}_k}{\hbar}\right)\Psi_k \\
&= \frac{i\Delta t\hbar}{\Delta x^2 m}(\Psi_{k+1}+\Psi_{k-1})-i\Delta t\left(\frac{\hbar}{\Delta x^2 m}+\frac{\mathcal{V}_k}{\hbar}\right)\Psi_k.
\end{aligned}$$

Next, to make the expression more concise, we introduce the following new quantities to use as abbreviations:

$$\epsilon = \frac{\hbar\Delta t}{m\Delta x^2},\ \omega_k = \frac{\mathcal{V}_k}{\hbar},\ \alpha_k = \epsilon + \omega_k\Delta t,$$

and rewrite our formula in terms of them:

$$\Delta_t \Psi_k = \frac{i\epsilon}{2}(\Psi_{k+1} + \Psi_{k-1}) - i\alpha_k \Psi_k.$$

But now, this is all just giving us $\Delta_t \Psi$—the change in Ψ over a time Δt. How exactly will this let us calculate Ψ at some future time given Ψ at the current time?

Well, using our earlier definition for Δ, we can expand the left hand side of the equation as follows (reintroducing t as an explicit argument):

$$\Psi_k(t + \frac{\Delta t}{2}) - \Psi_k(t - \frac{\Delta t}{2}) = \frac{i\epsilon}{2}(\Psi_{k+1}(t) + \Psi_{k-1}(t)) - i\alpha_k \Psi_k(t).$$

If we solve this for $\Psi_k(t + \Delta t/2)$, we get

$$\Psi_k(t + \frac{\Delta t}{2}) = \Psi_k(t - \frac{\Delta t}{2}) + \frac{i\epsilon}{2}(\Psi_{k+1}(t) + \Psi_{k-1}(t)) - i\alpha_k \Psi_k(t),$$

and if we now replace Δt everywhere with $2\Delta t$ (a change which does not matter since Δt was an arbitrarily chosen value already), including within the definitions of ϵ and α_k, we get

$$\Psi_k(t + \Delta t) = \Psi_k(t - \Delta t) + i\epsilon(\Psi_{k+1}(t) + \Psi_{k-1}(t)) - 2i\alpha_k \Psi_k(t).$$

Now for conciseness we replace $(t + \Delta t)$ as an argument with $s + 1$ as a subscript, similarly to what we did earlier when we replaced $(x + \Delta x)$ with $k + 1$, and obtain

$$\Psi_{k,s+1} = \Psi_{k,s-1} + i\left(\epsilon(\Psi_{k+1,s} + \Psi_{k-1,s}) - 2\alpha_k \Psi_{k,s}\right).$$

Well, there are a couple of important things to note about this equation. First rather than deriving the state of the wave at the next time step $s + 1$ from the state at step s, the equation requires using the states at the two previous steps s and $s - 1$. It's a second order difference equation, and it's of the form that Fredkin has shown to always be totally reversible, given a numeric representation that supports a reversible addition operation.

The second important point is that although the equation is complex, the real part of Ψ_{s+1} depends only on the real part of Ψ_{s-1} and the imaginary part of Ψ_s. Specifically:

$$\Re\Psi_{k,s+1} = \Re\Psi_{k,s-1} + \Im\left(2\alpha_k \Psi_{k,s} - \epsilon(\Psi_{k+1,s} + \Psi_{k-1,s})\right) \tag{E.3}$$

and similarly, the imaginary part of Ψ_{s+1} depends only on the imaginary part of Ψ_{s-1}

and the real part of Ψ_s:

$$\Im\Psi_{k,s+1} = \Im\Psi_{k,s-1} - \Re\left(2\alpha_k\Psi_{k,s} - \epsilon(\Psi_{k+1,s} + \Psi_{k-1,s})\right) \tag{E.4}$$

Therefore, if we consider the real components of Ψ at all the even-numbered steps $s = 2n$ (for n an integer), and the imaginary components of Ψ at all odd-numbered steps $s = 2n + 1$, we see that the evolution of those components is totally self-contained, and we can ignore the real components at odd times, and the imaginary components at even times, and thus work with only a single real number at each time step. Let us do so, and define, for integers n, the real quantities

$$\begin{aligned} \mathcal{X}_{k,n} &\equiv \Re\Psi_{k,2n} \\ \mathcal{Y}_{k,n} &\equiv \Im\Psi_{k,2n+1}. \end{aligned}$$

Then we can rewrite the equations (E.3 and E.4) above as two equations:

$$\begin{aligned} \mathcal{X}_{k,n+1} &= \mathcal{X}_{k,n} + f_k(\vec{\mathcal{Y}}_n) \\ \mathcal{Y}_{k,n+1} &= \mathcal{Y}_{k,n} - f_k(\vec{\mathcal{X}}_{n+1}), \end{aligned} \tag{E.5}$$

where (for $\mathcal{Q}$ being either $\mathcal{X}$ or $\mathcal{Y}$) by $\vec{\mathcal{Q}}_n$ we mean the vector of all values $\mathcal{Q}_{k,n}$, and where (now omitting the n subscript):

$$f_k(\vec{\mathcal{Q}}) = 2\alpha_k\mathcal{Q}_k - \epsilon(\mathcal{Q}_{k+1} + \mathcal{Q}_{k-1}).$$

The update rule (E.5) lends itself to a particularly simple pseudo-code implementation given reversible addition/subtraction instructions, as are the C language's `+=` and `-=` operators when performed on integers:

$$\begin{aligned} \vec{\mathcal{X}} &\mathrel{+}= \lfloor f(\vec{\mathcal{Y}})\rfloor \\ \vec{\mathcal{Y}} &\mathrel{-}= \lfloor f(\vec{\mathcal{X}})\rfloor, \end{aligned}$$

where the absence of the k subscript is intended to suggest application of each operation to every element of the given vector. The n is implicit, and is no longer needed; each value of $\vec{\mathcal{X}}$ that is computed will implicitly represent the value of $\vec{\mathcal{X}}$ two time steps beyond the previous value that was computed, and each value of $\vec{\mathcal{Y}}$ that is computed will implicitly represent the value of $\vec{\mathcal{Y}}$ one time step beyond the previous value of $\vec{\mathcal{X}}$, and two time steps beyond the previously computed $\vec{\mathcal{Y}}$.

This process of updating $\vec{\mathcal{X}}$ and $\vec{\mathcal{Y}}$, can then be exactly undone by:

$$\begin{aligned} \vec{\mathcal{Y}} &\mathrel{+}= \lfloor f(\vec{\mathcal{X}}) \rfloor \\ \vec{\mathcal{X}} &\mathrel{-}= \lfloor f(\vec{\mathcal{Y}}) \rfloor. \end{aligned}$$

So the above analysis gives us a method of reversibly updating two arrays, representing the real and imaginary parts of Ψ at successive times. Note that this does not tell us both components of Ψ at either particular time, but if Ψ is changing gradually, as it will be if Δt is sufficiently small, then the two components taken together will be a fairly accurate representation of Ψ's complex value at either of the times.

We note that if there is actually only one spatial point $k = 0$, so that the "neighboring" points $k+1$ and $k-1$, are actually, in modulus 1 arithmetic, the same point as k itself, then the above algorithm actually reduces to (letting $X = \mathcal{X}_0, Y = \mathcal{Y}_0$):

$$\begin{aligned} X &\mathrel{+}= \lfloor 2\omega_0 \Delta t Y \rfloor \\ Y &\mathrel{-}= \lfloor 2\omega_0 \Delta t X \rfloor, \end{aligned}$$

which is an approximate circle-drawing algorithm, with the X's and Y's giving the coordinates of points on (or near) a circle. It is essentially the same algorithm as that described by Margolus in [93], §2.8.2, pp. 82–84, with our energy-based quantity $2\omega_0\Delta t = 2\mathcal{V}_0\Delta t/\hbar$ representing a quantity which can be thought of as being approximately the amount of change in the (X, Y) vector per $2\Delta t$ time, as a fraction of its length. This quantity takes the place of the $2\sin(\omega)$ quantity which plays the same role in Margolus's equation, determining there the angular change (in radians) of the point (x_t, y_t) across a span of 2 time steps. Note that if $\Delta t = 1$ and $\omega \approx 0$, then $2\omega\Delta t \approx 2\sin(\omega)$.

Improved algorithms. The above algorithm has the flaw that if Δt is too large, so that ϵ and all α_ks do not all remain small numbers, then the resulting evolution will be far from the sort of unitary, norm-preserving operation that we would like to have.

The dynamical system that Margolus originally described did not suffer from this problem, since his was based on an iteration $z_{t+1} = e^{i\omega} z_t$ that was exact even if ω was large. We would really prefer to have a way of stepping through the Schrödinger equation that benefited from a similar property.

I have produced other, slightly more sophisticated versions of the above algorithm, which attempt to do a better job of preserving unitarity, at least, even when Δt is large.

Actually I think no algorithm based on discrete spatial simulation will work in the case where the potential energy function that is imposed on the system leads to a particle acquiring a momentum that corresponds to a wavelength that is small compared to the separation between points. However, as long as the separation between points is much smaller than the shortest wavelength that ever appears, it should be possible to construct a simulation that is reasonably accurate even when fairly large amounts of time are jumped over in a single step.

I could at this point go on and discuss in much more detail all my different variations of this simulation, and their pros and cons, but at this point it seems to be a low priority. For now, suffice it to say that I have a 100 percent reversible technique for simulating the evolution of the Schrödinger wave function in this simple system, it is accurate as a first order approximation (although I have not here taken the space to formalize and prove that assertion), and empirical demonstrations (on a normal computer) verify that the simulation behaves quite well in a variety of simple test cases involving different initial position distributions, velocities, and potential energy functions. Phenomena such as tunneling and interference have been observed. Total probability is nearly conserved. And total reversibility has been experimentally validated.

For Fredkin and Barton's update rule, which is essentially the same as ours, Richard Feynman apparently discovered that there *is* is a definition of total probability that is *exactly* conserved by the update rule [63]. In the context of our discussion, the corresponding definition of the exactly conserved probability over time steps n would be: $P_n = \vec{\mathcal{X}}_n^2 - \vec{\mathcal{Y}}_{n-1} \cdot \vec{\mathcal{Y}}_{n+1}$. We just recently learned of Feynman's invariant, and we have not yet checked our own update rule to make sure that it works.

We now show how to port the above algorithm to a reversible processor. Updating the state need involve only integer addition and multiplication. (Pendulum does not currently support a built-in multiplication operation, so I had to implement multiplication as a subroutine, which was easy to write in my high-level language.)

I believe this program, as it stands, constitutes an interesting demonstration of a significant reversible program in our reversible language, and also demonstrates the ability of a totally reversible program written in our language to simulate physics without incurring an asymptotically increasing need for storage.

E.2 Reversible C implementation

I have several different versions of my C program for simulating the Schrödinger equation. The following version, `schii.c`, is an exactly-reversible version that uses only integer arithmetic, and thus was the basis for the version of the program to run on Pendulum, since we have not created any floating-point support for Pendulum as of yet.

Large parts of this program are simply concerned with drawing the graphics display using the X window system, and are therefore uninteresting from the point of view of the simulation technique itself. We have tried to isolate most of the graphics code into a section at the bottom of the program.

The user interface to the program is currently very minimal. Any key press prints out current statistics about the wave function. Any mouse button press exits the program. To change any parameters of the simulation, one must edit the appropriate constant and recompile the program. Fortunately, the program is short enough so that this does not take very long.

The key functions in the program are: `signed_mult_frac()`, which is the integer multiplication routine for integers taken as representing fractions between -1 and 1, `function()`, which computes the appropriate function at a given point that gives the amount by which a component of the wave vector should be changed at that point, and `step_forwards()` and `step_back()`, which perform a state update in the given time direction. These functions contain the core functionality which we ported to R and Pendulum assembly.

```
/* SCHII: Like SCHI2 except uses only integer multiplication in the main loop.
   SCHI2 kept its state in integers but calculating amounts to add
   using floating-point math.

   This will be the model for my Pendulum implementation.
   */

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

/* Physical constants. We'll use MKS (m,kg,s,nt,J,coul...) units. */
#define planck_h (6.626e-34) /* Planck's constant, in Joule-seconds */
#define light_c (2.998e8) /* Speed of light in meters/second */
static const double
  hbar = planck_h/(2*M_PI),/* h/2*pi, also in J-sec */
  elec_m = 9.109e-31,/* Electron rest mass, in kg */
  elec_q = 1.602e-19,/* Electron charge (abs.value) in Coulombs */
  coul_const = 8.988e9; /* 1/4*pi*epsilon_0 (Coulomb's law constant)
   in nt-m^2/coul^2. */

/* Parameters of simulation.                                    o  */
#define space_width (1e-10) /* Width of sim space in meters: 1 A. */
#define num_pts (128) /* Number of discrete space points in sim. */
static const double
  sim_dx = space_width/num_pts, /* delta btw. pts, in meters. */
  sim_dt = 5e-22,/* Simulated time per step, in secs. */
  init_vel = light_c*0.0,/* Initial velocity in m/s */
  initial_mu = -space_width/4,/* initial mean electron pos, rel. to ctr. */
  initial_sigma = space_width/20; /* width of initial hump. */

/* For holding some arrays of size num_pts, in real/imaginary pairs. */
static double
  *energies,/* Real potential energies at points. */
  *on_real,*on_imag,
  *off_real,*off_imag,/* On/off diagonal matrix elements, real/imag */
  *Psi_real,*Psi_imag; /* Real at t, imag at t+1/2. */

/* Now, we will use Psi_real and Psi_imag only for translation between
   the internal, integer form, and how it is used externally.  Here is
   the real wave function: */

static int *psiR,*psiI; /* Real and imaginary integer wave function. */
static double scaleFactor; /* The value, in the integer range, that a real
   value of 1.0 translates to. */

static int n_steps = 0; /* Number of iterations done so far. */
static double total_t = 0; /* Total simulated time so far. */

static int max_steps = 10000; /* go a million iterations before reversing */
static int direction = 0; /* forwards */

#define STEPS_PER_SHOT 20

typedef enum energ_funcs {
  neg_gaus=0, pos_gaus, inv_cutoff, parabolic, const_nonzero, const_zero,
  step_barrier
} energ_func_id;

static energ_func_id which_potential = parabolic;

static const char *energ_func_strs[] = {
  "Negative Gaussian potential well",
  "Positive Gaussian potential bump",
  "Inverse distance well with cutoff",
  "Parabolic well for harmonic oscillator",
  "Constant, non-zero energy level",
  "Constant, zero energy",
  "A step barrier to tunnel through"
};

void print_sim_params() {
  printf("\n");
  printf("SCHROEDINGER SIMULATOR PARAMETERS\n");
  printf("---------------------------------\n");
  printf("\n");
  printf("Width of simulated space is %g meters (%g light-seconds).\n",
 space_width, space_width/light_c);
  printf("Simulating %d discrete points in space.\n", num_pts);
  printf("Distance between points: %g m (%g ls).\n",sim_dx,
 sim_dx/light_c);
  printf("Time per simulation step: %g secs (light dist: %g m)\n",
 sim_dt, sim_dt*light_c);
  printf("Initial electron position: mu=%g m, sigma=%g m.\n",
 initial_mu, initial_sigma);
  printf("Using potential energy function %d: %s.\n", which_potential,
 energ_func_strs[which_potential]);
  printf("Initial electron velocity = %g m/s (%g c).\n",
 init_vel, init_vel/light_c);
  printf("Number of steps to go before reversing: %d.\n",max_steps);
  printf("\n");
}

static double *init_real,*init_imag;

void print_stats() {
  /* Calculate and print some stats of the wavefunction. */
  int this = n_steps&1;
  int i;
  double total_p = 0;
  double mom_real = 0, mom_imag = 0,
    potential = 0,
    kin_real = 0, kin_imag = 0,
    energ_real = 0, energ_imag = 0;
  double dPsi2_real, dPsi2_imag;
  double diff=0;

  for(i=0;i<num_pts;i++){
    int next = (i+1)%num_pts,
      prev = (i-1+num_pts)%num_pts;
    double real = Psi_real[i],
      imag = Psi_imag[i];
    double pd = real*real+imag*imag;
    total_p += pd;

    dPsi2_real = Psi_real[next] - Psi_real[prev];
    dPsi2_imag = Psi_imag[next] - Psi_imag[prev];
    mom_real += real*dPsi2_imag - imag*dPsi2_real;
    mom_imag -= real*dPsi2_real + imag*dPsi2_imag;

    potential += pd*energies[i];
  }
  mom_real *= hbar/(2*sim_dx);
  mom_imag *= hbar/(2*sim_dx);

  for(i=0;i<num_pts;i++){
    double dr = Psi_real[i] - init_real[i];
    double di = Psi_imag[i] - init_imag[i];
    diff += dr*dr + di*di;
  }
  diff /= num_pts;

  printf("Cycle = %d, t = %g. Total P = %g.\n",n_steps,total_t,total_p);
  printf("   Mean squared diff. from init. state = %g. (RMS = %g)\n",
 diff, sqrt(diff));
  printf("   Momentum = (%g + i %g) kg m/s.\n",mom_real,mom_imag);
  printf("   Velocity = (%g + i %g) m/s.\n",
 mom_real/elec_m,mom_imag/elec_m);
  printf("            = (%g + i %g) c.\n",
 mom_real/(elec_m*light_c),mom_imag/(elec_m*light_c));
  printf("  Potential = %g J (%g eV)\n", potential, potential/1.602e-19);
}

/* Gaussian (normal) distribution, non-normalized. */
double normal(double x,double mu,double sigma)
{
  double d = (x-mu)/sigma;
  return exp(-0.5*d*d);
}

double *malloc_doubles(){
  return (double *)calloc(num_pts,sizeof(double));
}

int *malloc_ints(){
  return (int *)calloc(num_pts,sizeof(int));
}
```

```
/* x should now be a real position in space */
double energy(double x){
  switch(which_potential){
  case neg_gaus:
    /* Negative Gaussian potential well. */
    return - 3e-15 * normal(x,0,space_width/10);
  case pos_gaus:
    /* Positive Gaussian potential bump. */
    return 5e-15 * normal(x,0,space_width/10);
  case inv_cutoff:
    /* Well where energy drops with inverse distance from center, down
       to a cutoff threshold. */
    {
      double ax = (x<0?-x:x);
      double p;
      p = - (coul_const*(elec_q*elec_q)/ax);
      if (p > -4e-14) {
return p;
      } else {
return -4e-14;
      }
    }
  case parabolic:
    /* Parabolic well for harmonic oscillator. */
    return x*x*1e+6;
  case const_nonzero:
    /* Constant, nonzero energy.  Apparent wave rotation rate differs
       from zero-energy case.*/
    return -24e-15;
  case const_zero:
    /* Constant, zero energy. */
    return 0;
  case step_barrier:
    /* A step barrier through which to tunnel. */
    if (x < space_width * 0.15) {
      return 0;
    } else if (x < space_width * 0.18) {
      return 1e-15;
    } else {
      return 0;
    }
  }
  return 0;
}

static double epsilon;
static double *alphas;
static int *alphasi;
static int epsiloni;

void cache_energies() {
  int i;
  /* epsilon: an angle that roughly indicates how much of a
     point's amplitude gets spread to its neighboring points per time
     step.  A function of sim dt and dx parameters only. */
  epsilon = hbar*sim_dt/(elec_m*sim_dx*sim_dx);
  epsiloni = (int)((unsigned)(0x80000000) * epsilon);
  printf("Sim epsilon angle: %g radians (%g of a circle).\n",
 epsilon, epsilon/(2*M_PI));
  printf("Integer epsilon: %d\n",epsiloni);
  energies = malloc_doubles();
  on_real = malloc_doubles();
  on_imag = malloc_doubles();
  off_real = malloc_doubles();
  off_imag = malloc_doubles();
  alphas = malloc_doubles();
  alphasi = malloc_ints();
  printf("Integer alphas:\n");
  for(i=0;i<num_pts;i++){
    double x = (i - num_pts/2.0)*sim_dx; /* Position in space. */
    double alpha;
    energies[i] = energy(x);
    /* alpha: at this particular point, what's the absolute phase rotation
       angle for on-diagonal.  Energy makes a contribution. */
    alpha = epsilon + energies[i]*sim_dt/hbar;
    /* A = exp(i*Atheta) */
    on_real[i] = cos(epsilon) * cos(-alpha);
    on_imag[i] = cos(epsilon) * sin(-alpha);
    /* What's the phase rotation for off-diag (neighbors). */
    off_real[i] = sin(epsilon) * cos(M_PI/2 - alpha);
    off_imag[i] = sin(epsilon) * sin(M_PI/2 - alpha);

    alphas[i] = alpha;
    alphasi[i] = (int)((unsigned)(0x80000000)*alphas[i]*2);
    printf("%d ",alphasi[i]);
  }
  printf("\n");
}

void init_wave() {
  double *probs = malloc_doubles();
  int i;
  double tprob;
  double lambda;
  Psi_real = malloc_doubles();
  Psi_imag = malloc_doubles();
  psiR = malloc_ints();
  psiI = malloc_ints();
  init_real = malloc_doubles();
  init_imag = malloc_doubles();
  tprob = 0;
  for(i=0;i<num_pts;i++){
    double x = (i - num_pts/2.0)*sim_dx;
    /* Unnormalized initial probability of finding electron here. */
    double p = normal(x,initial_mu,initial_sigma);
    probs[i] = p;
    tprob += p;
  }
  for(i=0;i<num_pts;i++){
    probs[i] /= tprob;
  }
  lambda = planck_h/(elec_m*init_vel);
  printf("Initial de Broglie wavelength is %g m (%g ls).\n",
 lambda,lambda/light_c);
  for(i=0;i<num_pts;i++){
    double x = (i - num_pts/2.0)*sim_dx;
    Psi_real[i] = sqrt(probs[i]) * cos(x/lambda*2*M_PI);
    Psi_imag[i] = sqrt(probs[i]) * sin(x/lambda*2*M_PI);
  }
  {
    double maxval = 0;
    double absval;
    for(i=0;i<num_pts;i++){
      absval = Psi_real[i];
      absval = (absval<0)?-absval:absval;
      if (absval>maxval) maxval=absval;
      absval = Psi_imag[i];
      absval = (absval<0)?-absval:absval;
      if (absval>maxval) maxval=absval;
    }
    printf("The maximum absolute value initially is: %g.\n",maxval);
    /* We'll scale our integers so that they can hold values up to
       almost twice the largest initial value before they incur
       an overflow. */
    scaleFactor = (1<<30)/maxval;
    printf("Therefore the scale factor will be: %g.\n",scaleFactor);
  }
  /* Convert to integers. */
  printf("Integer psis:\n");
  for(i=0;i<num_pts;i++){
    psiR[i] = Psi_real[i]*scaleFactor;
    psiI[i] = Psi_imag[i]*scaleFactor;
    printf("%d+%di ",psiR[i],psiI[i]);
  }
  printf("\n");
  /* Convert back to doubles for convenience. */
  for(i=0;i<num_pts;i++){
    Psi_real[i] = init_real[i] = psiR[i]/scaleFactor;
    Psi_imag[i] = init_imag[i] = psiI[i]/scaleFactor;
  }
}

void sim_init () {
  print_sim_params();
  cache_energies();
  init_wave();
  print_stats();
}

int signed_mult_frac(int m1,int m2)
{
  int pos,prod=0;
  unsigned int mask = 1<<31;
  int m1p=m1,m2p=m2;
  if (m1<0) m1p = -m1p;
  if (m2<0) m2p = -m2p;
  for(pos=1;pos<32;pos++){
    mask >>= 1;
    if (m1p&mask)
      prod += m2p>>pos;
  }
  if (m1<0) prod = -prod;
  if (m2<0) prod = -prod;
  return prod;
}

double function(int *vec,int i){
  int j,k;
  j = i+1; if (j==-1) j=num_pts-1; else if (j==num_pts) j=0;
  k = i-1; if (k==-1) k=num_pts-1; else if (k==num_pts) k=0;
  return
    signed_mult_frac(alphasi[i],vec[i])
    - signed_mult_frac(epsiloni,vec[j])
    - signed_mult_frac(epsiloni,vec[k]);
}

void step_forwards() {
  int i;
  for(i=0;i<num_pts;i++)
    psiR[i] += (int)(function(psiI,i));
  for(i=0;i<num_pts;i++)
    psiI[i] -= (int)(function(psiR,i));
  for(i=0;i<num_pts;i++){
    Psi_real[i] = psiR[i]/scaleFactor;
    Psi_imag[i] = psiI[i]/scaleFactor;
  }
  n_steps++;
  total_t+=sim_dt;
}

void step_back() {
  int i;
  n_steps--;
  for(i=0;i<num_pts;i++)
    psiI[i] += (int)(function(psiR,i));
  for(i=0;i<num_pts;i++)
    psiR[i] -= (int)(function(psiI,i));
  for(i=0;i<num_pts;i++){
    Psi_real[i] = psiR[i]/scaleFactor;
    Psi_imag[i] = psiI[i]/scaleFactor;
  }
  total_t-=sim_dt;
}

void sim_step() {
  if (direction == 0) {
    step_forwards();
  } else {
    step_back();
  }
  if (direction == 1 && n_steps == 0) {
    printf("\nPresumably back to initial state.\n");
    print_stats();
    printf("Now turning and going forwards.\n");
    direction = 0;
  } else if (direction == 0 && n_steps == max_steps) {
    printf("\nCompleted %d steps.\n", max_steps);
    print_stats();
    direction = 1;
    printf("Now turning around and going backwards.\n");
  }
}

/*----------------------------------------------------------------------*/
/* Graphics. */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include "icon.bitmap"
#define BITMAPDEPTH 1
static Display *display;
static int screen;
static double *prev_real, *prev_imag;

static double maxv;
static double maxp;

void init_graphics() {
  int i;
  int this = n_steps&1;
  prev_real = malloc_doubles();
  prev_imag = malloc_doubles();
  maxv = 0;
  maxp = 0;
  for(i=0;i<num_pts;i++){
    double absv;
    absv = Psi_real[i];
    absv = (absv<0)?-absv:absv;
    if (absv > maxv) maxv = absv;
    absv = Psi_imag[i];
    absv = (absv<0)?-absv:absv;
    if (absv > maxv) maxv = absv;
    absv = Psi_real[i]*Psi_real[i]
      + Psi_imag[i]*Psi_imag[i];
    if (absv > maxp) maxp = absv;
  }
}

void draw_graphics(win,gc,window_width,window_height,gce,gcreal,gcimag)
     Window win;
     GC gc;
     unsigned window_width, window_height;
     GC gce,gcreal,gcimag;
{
  double ght = window_height/2; /* How much Y space per graph */
  unsigned c1 = ght/2; /* origin y for top graph */
  unsigned c2 = window_height; /* origin y for bottom graph */
  int this = n_steps&1; /* which Psi is current */
  int i;

  for(i=0;i<num_pts;i++){
    unsigned x = i*window_width/num_pts;
    double prevReal = prev_real[i],
      prevImag = prev_imag[i],
      thisReal = Psi_real[i],
      thisImag = Psi_imag[i];
    double prevProb = prevReal*prevReal + prevImag*prevImag,
      thisProb = thisReal*thisReal + thisImag*thisImag;
    int prY = (int)((prevReal/maxv)*ght*0.5);
    int piY = (int)((prevImag/maxv)*ght*0.5);
    int ppY = (int)((prevProb/maxp)*ght);
    int trY = (int)((thisReal/maxv)*ght*0.5);
    int tiY = (int)((thisImag/maxv)*ght*0.5);
    int tpY = (int)((thisProb/maxp)*ght);

    /* Similar to above calculation, but for point next to us on the right.*/
    unsigned j = (i+1)%num_pts;
    unsigned xj = (i+1)*window_width/num_pts;
    double RprevReal = prev_real[j],
      RprevImag = prev_imag[j],
      RthisReal = Psi_real[j],
      RthisImag = Psi_imag[j];
    double RprevProb = RprevReal*RprevReal + RprevImag*RprevImag,
      RthisProb = RthisReal*RthisReal + RthisImag*RthisImag;
    int RprY = (int)((RprevReal/maxv)*ght*0.5);
    int RpiY = (int)((RprevImag/maxv)*ght*0.5);
    int RppY = (int)((RprevProb/maxp)*ght);
    int RtrY = (int)((RthisReal/maxv)*ght*0.5);
    int RtiY = (int)((RthisImag/maxv)*ght*0.5);
    int RtpY = (int)((RthisProb/maxp)*ght);

    /* Erase old Psi at this pos. */
    XDrawLine(display,win,gce,x,c1-prY,xj,c1-RprY);
    XDrawLine(display,win,gce,x,c1-piY,xj,c1-RpiY);
    /* Draw new Psi. */
    XDrawLine(display,win,gcreal,x,c1-trY,xj,c1-RtrY);
    XDrawLine(display,win,gcimag,x,c1-tiY,xj,c1-RtiY);
    /* Erase old prob and draw new. */
    XDrawLine(display,win,gce,x,c2-ppY,xj,c2-RppY);
    XDrawLine(display,win,gc,x,c2-tpY,xj,c2-RtpY);
    /* Draw energy function. */
    XDrawPoint(display,win,gc,x,(int)(c2-(ght*0.5)-ght*energies[i]*1e14));
  }
  for(i=0;i<num_pts;i++){
    prev_real[i] = Psi_real[i];
    prev_imag[i] = Psi_imag[i];
  }
  XFlush(display);
}

/*----------------------------------------------------------------------*/
/* Pretty much everything below here is uninteresting X interfacing
   stuff. */

get_GC(Window win, GC *gc, XFontStruct *font_info, int foo) {
  unsigned long valuemask = 0; /* ignore XGCvalues and use defaults */
  XGCValues values;
  unsigned int line_width = 1;
  int line_style = LineSolid;
  int cap_style = CapButt;
  int join_style = JoinRound;
  int dash_offset = 0;
  static char dash_list[] = {
    12, 24 };
  int list_length = 2;

  /* Create default graphics context */
  *gc = XCreateGC(display,win,valuemask,&values);

  /* specify font */
  XSetFont(display,*gc,font_info->fid);

  {
    XColor sdr,edr;
  if (foo == 1) {
    XSetForeground(display,*gc,WhitePixel(display,screen));
  } else if (foo == 0) {
    XSetForeground(display,*gc,BlackPixel(display,screen));
  } else if (foo == 2) {
    XAllocNamedColor(display,DefaultColormap(display,screen),"cyan",
&sdr,&edr);
    XSetForeground(display,*gc,edr.pixel);
  } else if (foo == 3) {
    XAllocNamedColor(display,DefaultColormap(display,screen),"yellow",
&sdr,&edr);
    XSetForeground(display,*gc,edr.pixel);
  }}

  /* set line attributes */
  XSetLineAttributes(display,*gc,line_width,line_style,cap_style,
     join_style);

  /* set dashes to be line_width in length */
  XSetDashes(display,*gc,dash_offset,dash_list,list_length);
}

load_font(XFontStruct **font_info) {
  char *fontname = "9x15";

  /* Access font */
  if ((*font_info = XLoadQueryFont(display,fontname)) == NULL) {
    fprintf(stderr,"Basic: Cannot open 9x15 font\n");
    exit(-1);
  }
}

int main(argc,argv)
     int argc;
     char **argv;
{
  Window win;
  unsigned width, height; /* window size */
  int x = 0, y = 0; /* window position */
  unsigned border_width = 4; /* border four pixels wide */
  unsigned display_width, display_height;
  char *window_name = "Schroedinger Wave Simulator";
  char *icon_name = "schroed";
  Pixmap icon_pixmap;
  XSizeHints size_hints;
  XEvent report;
  GC gc,gce,gcreal,gcimag;
  XFontStruct *font_info;
  char *display_name = NULL;
  int i;

  /* connect to X server */
  if ( (display=XOpenDisplay(display_name)) == NULL ) {
    fprintf(stderr,
    "cannot connect to X server %s\n",
    XDisplayName(display_name));
    exit(-1);
  }

  sim_init();
  init_graphics();

  /* get screen size from display structure macro */
  screen = DefaultScreen(display);

  display_width = DisplayWidth(display,screen);
  display_height = DisplayHeight(display,screen);

  /* size window with enough room for text */
  width = display_width/3, height = display_height/3;
  /* create opaque window */
  win = XCreateSimpleWindow(display,RootWindow(display,screen),
    x,y,width,height,border_width,
    WhitePixel(display,screen),
    BlackPixel(display,screen));

  /* Create pixmap of depth 1 (bitmap) for icon */
  icon_pixmap = XCreateBitmapFromData(display, win, icon_bitmap_bits,
      icon_bitmap_width,
      icon_bitmap_height);

  /* initialize size hint property for window manager */
  size_hints.flags = PPosition | PSize | PMinSize;
  size_hints.x = x;
  size_hints.y = y;
  size_hints.width = width;
  size_hints.height = height;
  size_hints.min_width = 175;
  size_hints.min_height = 125;

  /* set properties for window manager (always before mapping) */
  XSetStandardProperties(display,win,window_name,icon_name,
 icon_pixmap,argv,argc,&size_hints);

  /* Select event types wanted */
  XSelectInput(display,win, ExposureMask | KeyPressMask |
       ButtonPressMask | StructureNotifyMask);

  load_font(&font_info);

  /* create GC for text and drawing */
  get_GC(win, &gc, font_info, 1);
  get_GC(win, &gce, font_info, 0);
  get_GC(win, &gcreal, font_info, 2);
  get_GC(win, &gcimag, font_info, 3);

  /* Display window */
  XMapWindow(display,win);

  while (1) { /* Event loop. */
    int i;
    XNextEvent(display,&report);
    switch(report.type) {
    case Expose:
      /* get rid of all other Expose events on the queue */
      while (XCheckTypedEvent(display, Expose, &report));
      draw_graphics(win, gc, width, height, gce, gcreal, gcimag);
      for(i=0;i<STEPS_PER_SHOT;i++)
sim_step();
      /*print_stats();
sleep(1);*/
      XClearArea(display,win,0,0,1,1,1);
      break;
    case ConfigureNotify:
      width = report.xconfigure.width;
      height = report.xconfigure.height;
      XClearArea(display,win,0,0,width,height,1);
      break;
    case KeyPress:
      print_stats();
      break;
    case ButtonPress:
      XUnloadFont(display,font_info->fid);
      XFreeGC(display,gc);
      XFreeGC(display,gce);
      XFreeGC(display,gcreal);
      XFreeGC(display,gcimag);
      XCloseDisplay(display);
      exit(1);
    default:
      break;
    }
  }
  return 0;
}
```

E.3 Source code in R language

The following is the complete source code for the Schrödinger simulation as ported into the R programming language, except for the multiplication routine, which the compiler provides as a standard library function. See the `def-smf` construct in §D.4.16 (p. 348) for the R source for the multiplication subroutine.

Note that the initial wavefunction state is provided in the form of a static data array, so that we do not have to port the trigonometric functions that were used to generate the initial state in the C version of the program. Also note that we did not bother to port the graphics code. Output is instead provided in a raw form which is parsed, displayed, and compared with the original C program's output by a separate program which is wrapped around the Pendulum simulator.

```
;;;----------------------------------------------------------------------
;;;
;;;  Schroedinger Wave Equation simulation program.
;;;  The first major test of R (the reversible language)!
;;;
;;;  Current status: More compiler work needed. 6/12/97.
;;;
;;;----------------------------------------------------------------------

;;; Currently all data must come before the code that uses it, so that the
;;; compiler will recognize these identifiers as names of static data items
;;; rather than as dynamic variables.

;; epsilon = hbar*dt/m*dx^2.  DX=7.8125e-13m, DT=5e-22s
(defword epsilon 203667001) ; 0.0948398 radians.
;; Parabolic potential well with 128 points.
(defarray alphas
  458243442 456664951 455111319 453582544 452078627 450599569 449145369
  447716027 446311542 444931917 443577149 442247239 440942188 439661994
  438406659 437176182 435970563 434789802 433633899 432502854 431396668
  430315339 429258869 428227257 427220503 426238607 425281569 424349389
  423442068 422559605 421701999 420869252 420061363 419278332 418520159
  417786845 417078388 416394790 415736049 415102167 414493143 413908977
  413349669 412815220 412305628 411820895 411361019 410926002 410515843
  410130542 409770099 409434515 409123788 408837920 408576909 408340757
  408129463 407943027 407781450 407644730 407532868 407445865 407383720
  407346432 407334003 407346432 407383720 407445865 407532868 407644730
  407781450 407943027 408129463 408340757 408576909 408837920 409123788
  409434515 409770099 410130542 410515843 410926002 411361019 411820895
  412305628 412815220 413349669 413908977 414493143 415102167 415736049
  416394790 417078388 417786845 418520159 419278332 420061363 420869252
  421701999 422559605 423442068 424349389 425281569 426238607 427220503
  428227257 429258869 430315339 431396668 432502854 433633899 434789802
  435970563 437176182 438406659 439661994 440942188 442247239 443577149
  444931917 446311542 447716027 449145369 450599569 452078627 453582544
  455111319 456664951)

;; This is the shape of the initial wavefunction; amplitude doesn't matter.
;; Real part.
(defarray psiR 2072809 3044772 4418237 6333469 8968770 12546502 17338479
  23669980 31921503 42527251 55969298 72766411 93456735 118573819 148615999
  184009768 225068513 271948808 324607187 382760978 445857149 513053161
  583213481 654924586 726530060 796185813 861933650 921789572 973841548
  1016350163 1047844835 1067208183 1073741824 1067208183 1047844835
  1016350163 973841548 921789572 861933650 796185813 726530060 654924586
  583213481 513053161 445857149 382760978 324607187 271948808 225068513
  184009768 148615999 118573819 93456735 72766411 55969298 42527251
  31921503 23669980 17338479 12546502 8968770 6333469 4418237 3044772
  2072809 1393998 926112 607804 394060 252382 159681 99804 61622 37586
  22647 13480 7926 4604 2642 1497 838 463 253 136 73 38 20 10 5 2 1 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
;;Imaginary part.
(defarray psiI 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)

;; This subroutine updates one of the two waves, using the other.
(defsub halfstep (dest src)
  (let (e <- epsilon)
    (for i = 0 to 127
       (let (d <-> (dest _ i))
          (d += ((alphas _ i) */ (src _ i)))
          (d -= (e */ (src _ ((i + 1) & 127))))
          (d -= (e */ (src _ ((i - 1) & 127))))))))

;; Print the current wave to the output stream.
(defsub printwave (wave)
  (for i = 0 to 127
       (printword (wave _ i)))
  (println))

;; Main program, goes by the name of SCHROED.
(defmain schroed
  (for i = 1 to 1000 ;Enough time for electron to fall to well bottom.
     ;; Take turns updating the two components of the wave.
     (call halfstep psiR psiI)
     (rcall halfstep psiI psiR)
     ;; Print both wave components.
     (call printwave psiR)
     (call printwave psiI)))
```

E.4 Compiled PISA code

The following is the exact PISA assembly code output that was produced from the above input file by the R compiler RCOMP we listed in ch. D. It consists of 830 machine words (395 of data, 435 of program). We will not review this code in detail here. However, when it was executed under our Pendulum virtual machine PENDVM (a C program written by Matt DeBergalis), it was found to produce exactly identical output, on every step, to that of the original C version of the program, listed earlier, thus validating the correctness of RCOMP and PENDVM (at least for this program). And when execution was stopped and reversed at any point, the processor state returned exactly to the orignal state at the start of the program (validating PENDVM's guarantee of reversibility).

```
;; pendulum pal file
_PRESKIP395:    BRA _POSTSKIP396
EPSILON:        DATA 203667001
_POSTSKIP396:   BRA _PRESKIP395
_PRESKIP397:    BRA _POSTSKIP398
ALPHAS:         DATA 458243442
                DATA 456664951
                DATA 455111319

                  ⋮  (122 intervening data statements elided)

                DATA 453582544
                DATA 455111319
                DATA 456664951
_POSTSKIP398:   BRA _PRESKIP397
_PRESKIP399:    BRA _POSTSKIP400
PSIR:           DATA 2072809
                DATA 3044772
                DATA 4418237

                  ⋮  (122 intervening data statements elided)

                DATA 0
                DATA 0
                DATA 0
_POSTSKIP400:   BRA _PRESKIP399
_PRESKIP401:    BRA _POSTSKIP402
PSII:           DATA 0
                DATA 0
                DATA 0

                  ⋮  (122 intervening data statements elided)

                DATA 0
                DATA 0
                DATA 0
_POSTSKIP402:   BRA _PRESKIP401
_SUBTOP403:     BRA _SUBBOT404
```

```
HALFSTEP:       SWAPBR $2
                NEG $2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
                ADDI $31 EPSILON
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                EXCH $30 $31
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADD $29 $30
                EXCH $30 $31
                ADDI $31 -EPSILON
                ADDI $30 128
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                ADD $31 $28
_FORTOP407:     BNE $31 $28 _FORBOT408
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADD $3 $31
                EXCH $27 $3
                SUB $3 $31
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                ADDI $26 ALPHAS
                ADD $26 $31
                ADDI $1 -7
                EXCH $25 $1
                ADDI $1 7
                ADDI $1 -8
                EXCH $24 $1
                ADDI $1 8
                EXCH $24 $26
                ADD $25 $24
                EXCH $24 $26
                SUB $26 $31
                ADDI $26 -ALPHAS
                ADD $24 $4
                ADD $24 $31
                ADDI $1 -9
                EXCH $23 $1
                ADDI $1 9
                EXCH $23 $24
                ADD $26 $23
                EXCH $23 $24
                SUB $24 $31
                SUB $24 $4
                XOR $23 $5
                XOR $5 $23
                XOR $26 $4
                XOR $4 $26
                XOR $26 $4
                XOR $25 $3
                XOR $3 $25
XOR $25 $3
XOR $24 $2
XOR $2 $24
ADDI $1 -9
BRA SMF
ADDI $1 9
ADD $27 $5
ADDI $1 -9
RBRA SMF
ADDI $1 9
ADD $2 $26
ADD $2 $31
ADDI $1 -10
EXCH $22 $1
ADDI $1 10
EXCH $22 $2
SUB $4 $22
EXCH $22 $2
SUB $2 $31
SUB $2 $26
ADDI $2 ALPHAS
ADD $2 $31
EXCH $4 $2
SUB $3 $4
EXCH $4 $2
SUB $2 $31
ADDI $2 -ALPHAS
ADD $2 $31
ADDI $2 1
ADDI $3 127
ANDX $4 $2 $3
ADDI $3 -127
ADD $3 $26
ADD $3 $4
EXCH $22 $3
ADD $5 $22
EXCH $22 $3
SUB $3 $4
SUB $3 $26
XOR $3 $5
XOR $5 $3
XOR $3 $4
XOR $4 $3
XOR $3 $4
XOR $29 $3
XOR $3 $29
XOR $29 $3
XOR $22 $2
XOR $2 $22
ADDI $1 -10
BRA SMF
ADDI $1 10
SUB $27 $5
ADDI $1 -10
RBRA SMF
ADDI $1 10
ADD $2 $26
ADD $2 $29
ADDI $1 -11
EXCH $21 $1
ADDI $1 11
```

```
EXCH $21 $2
SUB $4 $21
EXCH $21 $2
SUB $2 $29
SUB $2 $26
ADDI $2 127
ANDX $29 $22 $2
ADDI $2 -127
ADDI $22 -1
SUB $22 $31
ADD $2 $31
ADDI $2 -1
ADDI $4 127
ANDX $5 $2 $4
ADDI $4 -127
ADD $4 $26
ADD $4 $5
EXCH $22 $4
ADD $21 $22
EXCH $22 $4
SUB $4 $5
SUB $4 $26
XOR $4 $5
XOR $5 $4
XOR $21 $4
XOR $4 $21
XOR $21 $4
XOR $22 $2
XOR $2 $22
ADDI $1 -11
BRA SMF
ADDI $1 11
SUB $27 $5
ADDI $1 -11
RBRA SMF
ADDI $1 11
ADD $2 $26
ADD $2 $21
EXCH $29 $2
SUB $4 $29
EXCH $29 $2
SUB $2 $21
SUB $2 $26
ADDI $2 127
ANDX $21 $22 $2
ADDI $2 -127
ADDI $22 1
SUB $22 $31
ADD $25 $31
EXCH $27 $25
SUB $25 $31
ADDI $31 1
ADDI $1 -11
EXCH $21 $1
ADDI $1 11
XOR $29 $3
XOR $3 $29
ADDI $1 -10
EXCH $22 $1
ADDI $1 10
XOR $2 $24
```

```
                XOR $24 $2
                XOR $3 $25
                XOR $25 $3
                XOR $4 $26
                XOR $26 $4
                XOR $5 $23
                XOR $23 $5
                ADDI $1 -9
                EXCH $23 $1
                ADDI $1 9
                ADDI $1 -8
                EXCH $24 $1
                ADDI $1 8
                ADDI $1 -7
                EXCH $25 $1
                ADDI $1 7
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
_FORBOT408:     BNE $31 $30 _FORTOP407
                SUB $31 $30
                ADDI $30 -128
                ADDI $28 EPSILON
                EXCH $30 $28
                SUB $29 $30
                EXCH $30 $28
                ADDI $28 -EPSILON
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
_SUBBOT404:     BRA _SUBTOP403
_SUBTOP444:     BRA _SUBBOT445
PRINTWAVE:      SWAPBR $2
                NEG $2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
                ADDI $31 128
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADD $30 $29
_FORTOP446:     BNE $30 $29 _FORBOT447
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                OUTPUT $28
                ADD $28 $3
                ADD $28 $30
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                EXCH $26 $28
                ADD $27 $26
                EXCH $26 $28
                SUB $28 $30
                SUB $28 $3
                OUTPUT $27
                ADD $26 $3
                ADD $26 $30
                EXCH $28 $26
                SUB $27 $28
                EXCH $28 $26
                SUB $26 $30
                SUB $26 $3
                ADDI $30 1
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
_FORBOT447:     BNE $30 $31 _FORTOP446
                SUB $30 $31
                ADDI $31 -128
                ADDI $29 1
                OUTPUT $29
                ADDI $29 -1
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
_SUBBOT445:     BRA _SUBTOP444
_SUBTOP457:     BRA _SUBBOT458
SMF:            SWAPBR $2
                NEG $2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                ADDI $29 1
                ADD $31 $3
_IFTOP459:      BGEZ $3 _IFBOT460
                NEG $31
_IFBOT460:      BGEZ $3 _IFTOP459
                ADD $30 $4
_IFTOP461:      BGEZ $4 _IFBOT462
                NEG $30
_IFBOT462:      BGEZ $4 _IFTOP461
                RL $29 31
                ADDI $1 -7
                EXCH $25 $1
                ADDI $1 7
                ADDI $25 1
                ADDI $1 -8
                EXCH $24 $1
                ADDI $1 8
                ADDI $24 32
                ADDI $1 -9
                EXCH $23 $1
                ADDI $1 9
                ADD $23 $25
_FORTOP463:     BNE $23 $25 _FORBOT464
                RR $29 1
                ANDX $27 $31 $29
_IFTOP467:      BEQ $27 $0 _IFBOT468
                SRLVX $28 $30 $23
                ADD $26 $28
                SRLVX $28 $30 $23
_IFBOT468:      BEQ $27 $0 _IFTOP467
                ANDX $27 $31 $29
                ADDI $23 1
_FORBOT464:     BNE $23 $24 _FORTOP463
                SUB $23 $24
                ADDI $24 -32
                ADDI $25 -1
_IFTOP469:      BGEZ $3 _IFBOT470
                NEG $26
_IFBOT470:      BGEZ $3 _IFTOP469
_IFTOP471:      BGEZ $4 _IFBOT472
                NEG $26
_IFBOT472:      BGEZ $4 _IFTOP471
                ADD $5 $26
_IFTOP473:      BGEZ $4 _IFBOT474
                NEG $30
_IFBOT474:      BGEZ $4 _IFTOP473
                SUB $30 $4
_IFTOP475:      BGEZ $3 _IFBOT476
                NEG $31
_IFBOT476:      BGEZ $3 _IFTOP475
                SUB $31 $3
                ADDI $29 -1
                ADDI $1 -9
                EXCH $23 $1
                ADDI $1 9
                ADDI $1 -8
                EXCH $24 $1
                ADDI $1 8
                ADDI $1 -7
                EXCH $25 $1
                ADDI $1 7
                ADDI $1 -6
                EXCH $26 $1
                ADDI $1 6
                ADDI $1 -5
                EXCH $27 $1
                ADDI $1 5
                ADDI $1 -4
                EXCH $28 $1
                ADDI $1 4
                ADDI $1 -3
                EXCH $29 $1
                ADDI $1 3
                ADDI $1 -2
                EXCH $30 $1
                ADDI $1 2
                ADDI $1 -1
                EXCH $31 $1
                ADDI $1 1
_SUBBOT458:     BRA _SUBTOP457
_MAINTOP:       BRA _MAINBOT
                .START SCHROED
SCHROED:        START
                ADDI $2 1
                ADDI $3 1001
                ADD $4 $2
_FORTOP477:     BNE $4 $2 _FORBOT478
                XOR $5 $4
                XOR $4 $5
                ADDI $4 PSII
                XOR $6 $3
                XOR $3 $6
                ADDI $3 PSIR
                XOR $7 $2
                XOR $2 $7
                BRA HALFSTEP
                ADDI $3 -PSIR
                ADDI $4 -PSII
                ADDI $4 PSIR
                ADDI $3 PSII
                RBRA HALFSTEP
                ADDI $3 -PSII
                ADDI $4 -PSIR
                ADDI $3 PSIR
                BRA PRINTWAVE
                ADDI $3 -PSIR
                ADDI $3 PSII
                BRA PRINTWAVE
                ADDI $3 -PSII
                ADDI $5 1
                XOR $2 $7
                XOR $7 $2
                XOR $3 $6
                XOR $6 $3
                XOR $4 $5
                XOR $5 $4
_FORBOT478:     BNE $4 $3 _FORTOP477
                SUB $4 $3
                ADDI $3 -1001
                ADDI $2 -1
                FINISH
_MAINBOT:       BRA _MAINTOP
```