September, 1998

PHY307F/407F - Computational Physics
Background Material for Expt. 2 - The Pendulum

David Harrison

INTRODUCTION

The usual way of writing the differential equation for a pendulum is:

$$l\frac{d^2\theta}{dt^2} + g\sin\theta = 0 \qquad (1)$$

where:

l = *length of the pendulum*
θ = *amplitude of oscillation*

g = *acceleration due gravity*

If the maximum amplitude is small, a good approximation is that $\sin\theta \approx \theta$, where θ is measured in radians. Equation (1) becomes:

$$l\frac{d^2\theta}{dt^2} + g\theta = 0$$

This equation is simple to solve. The solution is, of course, *simple harmonic motion*:

$$\theta = \theta_{max}\sin(\omega t + \phi)$$

where:

θ_{max} = *maximum amplitude*
$\omega = \sqrt{g/l}$
ϕ = *constant*

What is sometimes not made clear in introductory courses is that if we do not restrict ourselves to small maximum amplitudes, Equation (1) is *not* soluble in a closed analytic form. In this case, one may only approximate a solution.

Using computers allows us to find approximate solutions that are very close to correct in a reasonable period of time, and this experiment investigates techniques to solve the pendulum and other *ordinary differential equations (ODE)*. Such techniques are often call *numerical integration*.

These notes are organised as follows:

I. PHYSICS BACKGROUND

I.A The Hamiltonian Formulation

We will be solving the pendulum in a Hamiltonian formulation. Students who have already taken PHY351S - *Classical Mechanics* or an equivalent course will already know about this way of solving mechanics problems. This section is intended for students who do not already have this background.

The Hamiltonian formulation is just another way of writing Newton's laws of motion. For the case where the potential energy is not a function of the speed of the objects being described, the Hamiltonian H is just the total energy of the objects. It is usually written in terms of generalised momentum coordinates p and generalised position coordinates q. For the pendulum these coordinates are just:

$$q = \theta$$

$$p = ml\frac{d\theta}{dt} = ml\frac{dq}{dt}$$

We will work in a system of units where the mass m, acceleration due to gravity g, and the length l are all equal to one. Then the Hamiltonian is:

$$H = \frac{p^2}{2} + (1 - \cos q)$$

Hamilton's equations of motion, which are just another way of writing Newton's laws, are:

$$\frac{\partial H}{\partial q} = -\frac{dp}{dt}$$
$$\frac{\partial H}{\partial p} = \frac{dq}{dt}$$

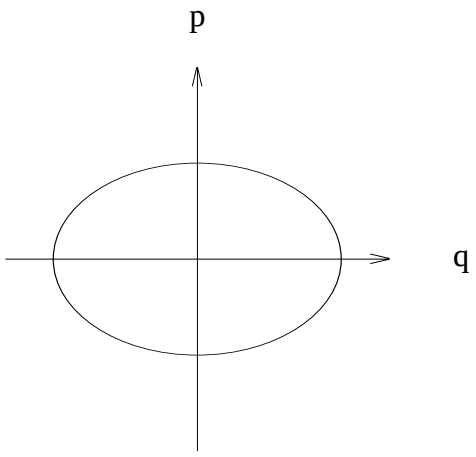
which for the pendulum becomes:

$$\frac{dp}{dt} = -\sin q \qquad \text{(I.1)}$$
$$\frac{dq}{dt} = p$$

It is these equations that we will be solving.

Often when using the Hamiltonian formulation one thinks in terms of *phase space*, which is essentially a graph of momentum versus position. This is the same phase space you may have encountered in, for example, a course in thermal physics.

For simple harmonic motion, such as the pendulum in the case of small maximum amplitude, the trajectory in phase space is an ellipse:



For the pendulum with arbitrary maximum amplitude the trajectory is approximately elliptical.

I.B Period of the Pendulum

As shown in elementary physics courses, in the case of small maximum amplitudes the period T of the pendulum is:

$$T_{simple} = 2\pi\sqrt{l/g} \tag{I.2}$$

Note that the period is independent of maximum amplitude in this approximation. For the system units we are using, with the acceleration due to gravity, and length of the pendulum both equal to one, T is equal to 2π.

For arbitrary maximum amplitude θ_{max}, although the solution of the equations of motion are not analytic, there are known analytic forms for the period. One form is:

$$T = 4\int_{\phi=0}^{\pi/2} \frac{d\phi}{\sqrt{1-\sin^2(\theta_{max}/2)\sin^2\phi}} \tag{I.3}$$

The integral is a complete elliptical integral of the first kind.

One may code *Mathematica* to calculate Equation (I.3):

```
period[Q_] := Module[ {k,phi},
     k = Sin[Q/2];
     4 * NIntegrate[Sqrt[ 1 /
          (1 - k^2 * Sin[phi]^2 ) ],
          {phi, 0, Pi/2}]
]
```

The above code will be supplied to you in the Notebook for the experiment.

There is another form for the period of the pendulum:

$$T = 2\sqrt{2}\int_{q=0}^{\theta_{max}} \frac{dq}{\sqrt{\cos q - \cos\theta_{max}}} \tag{I.4}$$

A *Mathematica* implementation of Equation I.4 will also be supplied to you in the Notebook for the experiment.

I.C The Fourier Transform

Most students in this course have been exposed to the Fourier transform in one way or another, but usually not in a very systematic way. This section reviews and possibly extends your knowledge of this transform.

Recall the Fourier series, in which a periodic function *f(t)* is written as a sum of sine and cosine terms:

$$f(t) = \sum_{n=1}^{\infty} a_n \cos(nt) + \sum_{n=1}^{\infty} b_n \sin(nt) \tag{I.5a}$$

or equivalently:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{-int} \tag{I.5b}$$

The coeffecients are found from the fact that the sine and cosine terms are orthogonal, from which:

$$a_n = \frac{1}{\pi} \int_{x=0}^{2\pi} f(x)\cos(nx)dx$$

$$b_n = \frac{1}{\pi} \int_{x=0}^{2\pi} f(x)\sin(nx)dx$$

Fourier series are used, for example, to discuss the harmonic structure of the tonic and overtones of a vibrating string.

Formally, the Fourier transform is similar to the above series, except that we integrate instead of summing the terms:

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{t=-\infty}^{\infty} f(t)e^{i\omega t}dt \tag{I.6}$$

Note that this integral transform replaces a function of, say, time *t* with another function of the angular frequency ω. Similarly, for a function of position, *f(x)*, the Fourier transform would replace it with a function of the wavenumber *g(k)*.

The inverse transform regenerates the original function from the transformed one:

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{\omega=-\infty}^{\infty} g(\omega)e^{-i\omega t}d\omega \tag{I.7}$$

The Fourier transform of a sine wave sin(ω*t*) is just a Dirac delta function centered at the frequency ω, which is just another way of saying the the Fourier decomposition has a single component of frequency ω.[1]

1. The Dirac delta function $\delta(t)$ is a function which is zero everywhere except at $t = 0$ and whose integral from $-\infty$ to ∞ is one. The mathematicians claim such a function does not exist, but in physics we use them all the time.

You will investigate Fourier transforms further in the preliminaries to Experiment 2.5, and then use the Fourier transform to do Experiment 2.5. To 'cut to the chase' you will discover that:

1. When one is taking a Fourier transform of, say, a few cycles of a sine wave instead of one that extends to infinity in both directions, "side bands" are produced in the spectrum.
2. When one is taking a Fourier transform of a sampled function instead of a continuous function, "aliases" of the transform are produced.

II. ALGORITHMS AND IMPLEMENTATION

II.A The Runge-Kutta Algorithm

In this experiment you will be investigating various types of Runge-Kutta algorithms. However, as Press et al. say:

> *For many scientific users, fourth-order Runge-Kutta is not just the first word on ODE [Ordinary Differential Equation] integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm .... Keep in mind, however, that the old workhorse's last trip may well take you to the poorhouse: Burlirsch-Stoer or predictor corrector methods can be very much more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields.*[2]

Although saying *Runge-Kutta* sounds pretty esoteric and possibly imposing, the idea is actually pretty simple. We are trying to solve an ordinary differential equation:

$$\frac{dx}{dt} \equiv x'(t) = f(x,t) \tag{II.1}$$

The initial condition $x(t)$ at time t, is known. Then at a time Δt later:

2. William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, **Numerical Recipes in C: The Art of Scientific Computing** (Cambridge Univ. Press), pg 571.

$$x(t + \Delta t) = x(t) + x'(t)\Delta t + \frac{1}{2}x''(t)(\Delta t)^2 + \cdots \qquad \text{(II.2)}$$

This is, of course, just a Taylor expansion.

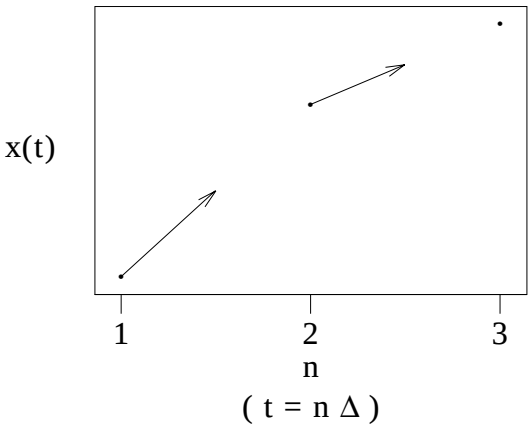
If we keep the first two terms in the expansion:

$$x(t + \Delta t) \equiv x_{n+1} \approx x_n + f(x_n, t_n)\Delta t \qquad \text{(II.3)}$$

Once we know x_{n+1}, we can find x_{n+2}:

$$x(t + 2\Delta t) \equiv x_{n+2} \approx x_{n+1} + f(x_{n+1}, t_{n+1})\Delta t \qquad \text{(II.4)}$$

In this way we can get approximate values for x at all times t. This is often called the *Euler method* of solving the differential equation; it is also known as a *first order Runge-Kutta*. If t is the time, we end up with a *time series* of values of *x(t)* at times $t = n\Delta$, $n = 1, 2, \ldots$. We can interpret the procedure graphically:



At each point, the derivative of *x(t)* with respect to *t* tells us in which direction to go to get to the next point in the graph.

A moment's reflection should convince you that this algorithm is pretty simple minded, probably too simple minded. If the slope of *f(x,t)* is, say, decreasing between *n* and *n+1*, then we will end up with a value of $x(t_{n+1})$ that is too large; similarly if the slope is increasing we will end up with a value that is too small. In principle we can increase the accuracy of the result be reducing the size of the time step Δt; you will investigate in the experiment why this sentence begins with the phrase *in principle*.

We can be more clever in deciding in which way to go to get to the next point in the time series by keeping the next term in the Taylor expansion:

$$x_{n+1} \approx x_n + f(x_n, t_n)\Delta t + \frac{1}{2}x''_n(t_n)(\Delta t)^2 \qquad \text{(II.5)}$$

It is reasonable to assume that there is some α between 0 and 1 for which:

$$x''(t) \approx \frac{x'(t + \alpha\Delta t) - x'(t)}{\alpha\Delta t} \qquad \text{(II.6)}$$

A bit of simple algebra yields:

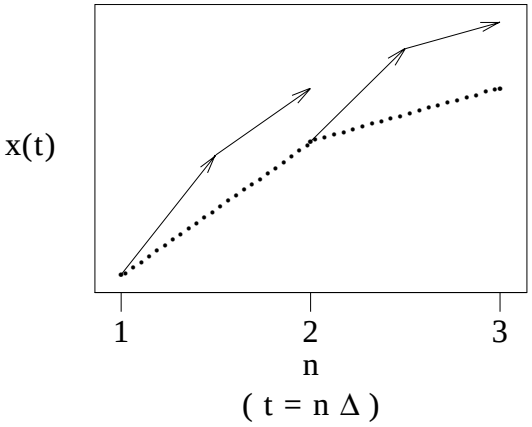
$$x_{n+1} = x_n + f(x_n, t_n)\Delta t + \frac{1}{2\alpha} f(\tilde{x}, t_n + \alpha\Delta t) \qquad \text{(II.7)}$$

where:

$\tilde{x} = x_n + f(x_n, t_n)\alpha\Delta t$

Say we choose $\alpha = 0.5$ in Equation (II.7). Then we can interpret the math to say:

1. Evaluate the slope at some time t_n. Move in that direction to the position $(t_{n+0.5}, x_{n+0.5})$.
2. Evaluate the slope f at that position.
3. Use that slope to determine how to move from time t_n to time t_{n+1}.



Note that there is no *a priori* reason for choosing α to be one half, or any other value.

The above procedure is a *second* order Runge-Kutta.

As one keeps additional terms in the Taylor expansion, one repeats the simplifications such as Equation (II.6). The interpretation is that as we add terms we are adding positions at which we evaluate the slope. At the end, for third and higher order Runge-Kuttas we average all these slopes to determine in which direction to go for the next term in the time series.

As we increase the order of the Runge-Kutta, we increase the accuracy of the solution for a given time step Δt. However, as we increase the order we also increase the amount of computation that we have to do. Although there is no known theoretical reason why it should be so, it turns out that for the typical ordinary differential equations that we encounter in physics the best tradeoff is the fourth order Runge-Kutta. In this scheme the slope is found at four positions:

$slope_1$: Evaluated at the point (t_n, x_n).

$slope_2$: Move in the direction given by $slope_1$ to time $t_{n+0.5}$, and evaluate the slope there.

$slope_3$: From the point (t_n, x_n) move in the direction given by $slope_2$ to time $t_{n+0.5}$ and evaluate the slope there.

$slope_4$: From the point (t_n, x_n) move in the direction given by $slope_3$ to time t_{n+1} and evaluate the slope.

Then an average of these four slopes is used to go from (t_n, x_n) to (t_{n+1}, x_{n+1}), where x_{n+1} is the next value of the time series.

Usually, the averaging is weighted as:

$$slope_1/6 + slope_2/3 + slope_3/3 + slope_4/6 \qquad \text{(II.8)}$$

There is no particular theoretical reason why this weighting is preferred, but a moment's thought may convince you that it is reasonable.

You should look at the C code listing in § IV of this document and/or the references to get further details on the various orders of Runge-Kuttas that you will be using in the experiment.

Finally, we close this sub-section with a brief discussion of the **symplectic** algorithm. Recall that in the Hamiltonian formulation we are finding the trajectory in *phase space*, which represents the motion of the pendulum. You may have learned in other courses that one usually thinks of dividing up phase space in "cells" of dimensions $\delta p \times \delta q$. Usually these cells are all of the same size, but the physics only demands that the areas of all the cells is the same.[3] The symplectic

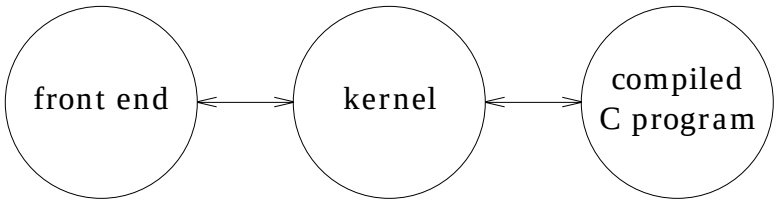
algorithm is based on a fourth order Runge-Kutta that has been modified to preserve areas in phase space.

We will treat the details of the symplectic algorithm as a "black box" and the contents of the box are somewhat beyond the scope of this course. The lesson that you will learn in doing the experiments, though, is that the symplectic algorithm is by far the best way to solve the pendulum. The general lesson, which will be re-enforced in later experiments, is that the physics demands a particular algorithm. Thus an expert in numerical methods who does not know the physics will choose algorithms that are very inferior to the choices that will be made by a physicist.

II.B Implementation

So far in this course, when you are working in the *Nortel* lab the *Mathematica* front end which provides the user interface is running on the PC you are using. The *Mathematica* kernel which does the actual work is running on *Faraday*, our central UNIX computer. Communication between the front end and the kernel is via the *MathLink* protocols, which were developed by Wolfram Research, the inventor of *Mathematica*.

Although it would be possible to code the Runge-Kutta algorithms used to solve the pendulum in the *Mathematica* language bundled into a package, as you have done for the Exercise and Experiment 1, this turns out to be too slow. Thus the actual Runge-Kutta's have been programmed in the C language. The C code is then compiled into a *binary* which links to the *Mathematica* kernel using the same *MathLink* protocols connecting the kernel to the front end. A diagram may clarify:



In addition to the C code itself, an accompanying *template* file defines the functions and user interface of the binary. You will be modifying the C code to produce Runge-Kutta algorithms of various orders and then recompiling to produce a

3. In Quantum Mechanics, one usually interprets this as the Heisenberg uncertainty principle, and the area is $\hbar$.

new binary. Listings of the template and C program appear in the § IV.

You will need to know that when the *Mathematica* kernel is connected to your compiled C program, the compiler refuses to replace the compiled program with a new one. Thus, you will need to disconnect the kernel from your program before you can compile you new C code into a binary. The Notebook for the experiment will tell you exactly how to do this.

III. REFERENCES

A paper describing the origins of this experiment is S.C. Douglas, D.M. Harrison and T.G. Shepherd, "The Physical Pendulum in an Advanced Undergraduate Course in Mechanics," Computers in Physics **8**, 416-419 (1995). The abstract and a PDF version of this paper may be found at

`http://faraday.physics.utoronto.ca/papers/papers.html`

III.A The Physics of the Pendulum

- H. Goldstein, **Classical Mechanics** (2nd ed.), Addison-Wesley. This is currently the textbook for PHY351S - *Classical Mechanics*.
- L.D. Landau and E.M. Lifshitz, **Mechanics**, Addison-Wesley. A classic text.
- I. Percival and D. Richards, **Introduction to Dynamics**, Cambridge. Formerly the text for PHY351S - *Classical Mechanics*.

III.B Runge Kutta algorithms

- Gene H. Golub and James M. Ortega, **Scientific Computing and Differential Equations** (Academic Press, 1992), § 2.1 - 2.2.
- William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, **Numerical Recipes: The Art of Scientific Computing or Numerical Recipes in C: The Art of Scientific Computing** (Cambridge Univ. Press), § 15.0 - 15.3.
- Djoko Wirosoetisno, "Symplectic Integration Algorithms". Written when he was a U of T undergraduate in April 1992, copies of this paper are available from David Harrison.

IV. CODE LISTINGS

IV.A C program

```
    /*
 * pendulum.c - the unperturbed physical pendulum
 *
```

```
 * This is the program that gets called from Mathematica via mathlink
 * when the mathematica Pendulum[] function (as defined in pendulum.tm)
 * is invoked.  Depending on the preprocessor macro ORDER, it will either
 * be a 1st-, 2nd-, 3rd-, 4th-, or 5th-order Runge-Kutta integrator, or
 * a 4th-order symplectic integrator.
 */

/* The next line is the Source Code Control System (SCCS) identification. */
static char SccsId[] = " @(#)pendulum.c   1.8 U of T Physics 06/19/93 ";

/*
 * ORDER is the order of the integration algorithm.  Set to 1, 2, 3, 4,
 * or 5 to get a Runge-Kutta scheme, or to SYMPLECTIC for a symplectic
 * algorithm.  To see the effect of setting different values, look at
 * function point() below.
 */

#define ORDER     4     /* This is intended to be edited! */

/* INCLUDE FILES: */

#include <math.h> /* For trigonometric functions, M_PI, etc. */
#include <stdio.h>      /* For fprintf */
#include "mathlink.h"   /* mathlink-specific definitions */

/* FUNCTIONS IN THIS FILE: */

void point();           /* The actual dirty work is done here */
double f();       /* the equations of motion */
void pendulum();  /* This is the C function called by mathlink */
int main();       /* Transfers control directly to mathlink */

/* PREPROCESSOR MACROS: */

/* SYMPLECTIC is a possible value of the ORDER macro */
#define SYMPLECTIC      -1
/* ham: the hamiltonian */
#define ham(q, p) ((p)*(p)/2.0 + (1.0 - cos(q)))

/* FUNCTION DEFINITIONS: */
```

```
/* point: this is the engine itself.
 *
 * q0:            the initial position
 * p0:            the initial momentum
 * pts:           how many points to return (note: the length of the array
 *          returned will be twice this if doubl is non-zero)
 * dt:            stepsize
 * doubl:   if this is non-zero, we'll return every point {q, p} as
 *          {q +- 2Pi, p} as well.  This is useful for the graphics but
 *          should not be chosen if further processing is going to be done
 *          on the data, e.g. a frequency spectrum or Liapunov exponent.
 */

void point(q, p, pts, dt, doubl)
double q, p, dt;
int doubl;
{
      int i, n = 0;

      /*
       * Declare some temporary variables.  How many of these are actually
       * declared depends on the integration scheme.
       */

#if (ORDER >= 1)
      double r[2];
#endif
#if (ORDER >= 2)
      double k1[2], k2[2];
#endif
#if (ORDER >= 3)
      double k3[2];
#endif
#if (ORDER >= 4)
      double k4[2];
#endif
#if (ORDER == 5)
      double k5[2], k6[2];
#endif

      while (n < pts) {
```

```
#if (ORDER == SYMPLECTIC) /* Symplectic integration algorithm */
/* cbrt2: the cube root of 2 */
#define cbrt2 1.259921049894873164767210607278228350570251464701 5
            q += (2+cbrt2+1/cbrt2)/6  * dt * f(q, p, 0);
            p += 1/(2-cbrt2)          * dt * f(q, p, 1);
            q += (1-cbrt2-1/cbrt2)/6  * dt * f(q, p, 0);
            p += 1/(1-cbrt2*cbrt2)    * dt * f(q, p, 1);
            q += (1-cbrt2-1/cbrt2)/6  * dt * f(q, p, 0);
            p += 1/(2-cbrt2)          * dt * f(q, p, 1);
            q += (2+cbrt2+1/cbrt2)/6  * dt * f(q, p, 0);

#endif
#if   (ORDER == 1) /* Also called "Euler's method". */
            for(i = 0; i < 2; i++)
                  r[i] = dt * f(q, p, i);

#endif
#if   (ORDER == 2) /* Simple mid-point rule */
            for(i = 0; i < 2; i++)
                  k1[i] = dt * f(q, p, i);
            for(i = 0; i < 2; i++) {
                  k2[i] = dt * f(q + 0.5 * k1[0], p + 0.5 * k1[1], i);
                  r[i] = k2[i];
            }

#endif
#if   (ORDER == 3) /* Similar to Simpson's rule */
            for(i = 0; i < 2; i++) {
                  k1[i] = dt * f(q, p, i);
                  r[i] = k1[i] / 6;
            }
            for(i = 0; i < 2; i++) {
                  k2[i] = dt * f(q + 0.5 * k1[0], p + 0.5 * k1[1], i);
                  r[i] += 2 * k2[i] / 3;
            }
            for(i = 0; i < 2; i++ ) {
                  k3[i] = dt * f(   q - k1[0] + 2 * k2[0],
                                    p - k1[1] + 2 * k2[1],
                                    i);
                  r[i] += k3[i] / 6;
            }
```

```
#endif
#if   (ORDER == 4) /* The standard choice for Runge-Kutta integration */
            for (i = 0; i < 2; i++) {
                  k1[i] = dt * f(q, p, i);
                  r[i] = k1[i] / 6;
            }
            for (i = 0; i < 2; i++) {
                  k2[i] = dt * f(q + 0.5 * k1[0], p + 0.5 * k1[1], i);
                  r[i] += k2[i] / 3;
            }
            for (i = 0; i < 2; i++) {
                  k3[i] = dt * f(q + 0.5 * k2[0], p + 0.5 * k2[1], i);
                  r[i] += k3[i] / 3;
            }
            for (i = 0; i < 2; i++) {
                  k4[i] = dt * f(q + k3[0], p + k3[1], i);
                  r[i] += k4[i] / 6;
            }

#endif
#if   (ORDER == 5) /* Note it takes 6 constants for 5th order! */
            for (i = 0; i < 2; i++) {
                  k1[i] = dt * f(q, p, i);
                  r[i] = k1[i] / 24;
            }
            for (i = 0; i < 2; i++)
                  k2[i] = dt * f(q + 0.5 * k1[0], p + 0.5 * k1[1], i);
            for (i = 0; i < 2; i++ ) {
                  k3[i] = dt * f( q + 0.25 * (k1[0] + k2[0]),
                              p + 0.25 * (k1[1] + k2[1]),
                              i);
            }
            for (i = 0; i < 2; i++ ) {
                  k4[i] = dt * f( q - k2[0] + 2 * k3[0],
                              p - k2[1] + 2 * k3[1],
                              i);
                  r[i] += 5 * k4[i] / 48;
            }
            for (i = 0; i < 2; i++ ) {
                  k5[i] = dt * f(
                    q + (7 * k1[0] + 10 * k2[0] + k4[0])/27,
```

```
                  p + (7 * k1[1] + 10 * k2[1] + k4[1])/27,
                  i);
            r[i] += 27 * k5[i] / 56;
      }
      for (i = 0; i < 2; i++ ) {
            k6[i] = dt * f(
              q + (28 * k1[0] + 546 * k3[0] + 54 * k4[0]
                  - 378 * k5[0])/ 625 - 0.2 * k2[0],
              p + (28 * k1[1] + 546 * k3[1] + 54 * k4[1]
                  - 378 * k5[1])/ 625 - 0.2 * k2[1],
              i);
            r[i] += 125  * k6[i] / 336;
      }
#endif      /* ORDER */

      /* Print a nice little dot on the screen */
      50*n % pts || fprintf(stderr, ".");

      n++;

#if (ORDER != SYMPLECTIC)     /* If the symplectic algorithm was used, */
      q += r[0];  /* q and p have already been incremented.  */
      p += r[1];
#endif

      /* "Wrap around" if q leaves desired range */
      q += M_PI;
      while (q < 0.0)
            q += 2 * M_PI;
      q = fmod(q, 2 * M_PI);
      q -= M_PI;

      /* Output the data point */
      MLPutFunction(stdlink, "List", 3);
      MLPutReal(stdlink, q);
      MLPutReal(stdlink, p);
      MLPutReal(stdlink, ham(q, p));

      /* Create the image, 2Pi away... */
      if (doubl) {
            MLPutFunction(stdlink, "List", 3);
```

```
                if (q < 0.0)
                      MLPutReal(stdlink, q + 2 * M_PI);
                else
                      MLPutReal(stdlink, q - 2 * M_PI);
                MLPutReal(stdlink, p);
                MLPutReal(stdlink, ham(q, p));
          }

      }
      fprintf(stderr,"\n");

}

/*
 * f: returns the appropriate derivative of the Hamiltonian.
 * i determines which derivative to return.
 */

double f(q, p, i)
double q, p;
int i;
{
      switch(i) {
            case 0: return p;
            case 1: return -sin(q);
            default:
                  fprintf(stderr, "Bad i (%s line %d)\n",
                                          __FILE__, __LINE__);
                  exit(1);
      }
      /*NOTREACHED*/
}


/* pendulum: the function called by mathlink when Pendulum is called in
 *         mathematica.
 *
 * Its arguments are the same as those of point().
 */

void pendulum(q0, p0, pts, dt, doubl)
```

```
double q0, p0, dt;
int doubl;
{
      fprintf(stderr, "pendulum(%g, %g, %d, %g, %d);\n",
                                    q0, p0, pts, dt, doubl);

      MLPutFunction(stdlink, "List", (1 + !!doubl) * pts);
            point(q0, p0, pts, dt, doubl);
      MLEndPacket(stdlink);
}


/* main: this simply transfers control to mathlink.
 */

int main()
{
      MLMain();
      return 0;
}
```

IV.B Template

```
     :: pendulum.tm
::
:: @(#)pendulum.tm      1.6 U of T Physics 10/08/96
::
:: This is the mathlink template file for the skeleton pendulum program.
:: It specifies the pattern for the Mathematica function Pendulum[] and the
:: way in which the Pendulum[] arguments will be translated into arguments
:: to the C function pendulum().
::
:: The function can be called from Mathematica in these ways:
::
::    Pendulum[{q0, p0}, {pts, dt}, double]
::    Pendulum[{q0, p0}, {pts, dt}]
::
:: {q0, p0} are the initial conditions; pts is the number of points to
:: return; dt is the timestep length, double is either True or False depending
:: on whether you want the phase portrait to cover -2Pi < q < 2Pi or just
:: -Pi < q < Pi, respectively.  The default value of double is False.
```

```
:Begin:

:: Specify what the name of the C function to be called is:

:Function:  pendulum

:: Define what the arguments to Pendulum[] are and what patterns they must
:: match.  (The contortions with Im[] etc. are necessary to allow any real
:: numbers.  See page 238 of the Mathematica book.  The N[] causes
:: expressions such as Pi/2 to be converted to numeric form in order that
:: NumberQ[] return True.)

:Pattern:   Pendulum[
                  point0:{
                        (q0_ /; NumberQ[N[q0]] && Im[N[q0]]==0),
                        (p0_ /; NumberQ[N[p0]] && Im[N[p0]]==0)
                  },
                  time:{
                        (pts_Integer /; pts > 0),
                        (dt_ /; NumberQ[N[dt]] && Im[N[dt]]==0)
                  },
                  (double_ /; double === True || double === False)
            ]

:: Then define what the arguments to the C function will be.  All the reals
:: are converted to C doubles, the integer will be sent as is, and double
:: (which has a boolean value) is converted to an integer with If.

:Arguments: { N[q0, 25], N[p0, 25], pts, N[dt, 25], If[double, 1, 0] }

:ArgumentTypes:   { Real, Real, Integer, Real, Integer }

:: The return value of Pendulum[] will not be the return value of the C
:: function but will instead be explicitly sent down the link, in other
:: words sent manually.  This is because there is no way of returning a
:: Mathematica list from a C function.

:ReturnType:      Manual

:End:
```

```
:: Set up an alternate calling sequence for the Mathematica function:

:Evaluate:  Pendulum[point0_, time_] := Pendulum[point0, time, False]

:: A help message so that you can type "?Pendulum" from within Mathematica
:: to get information about the function:

:Evaluate:  Pendulum::usage =
      "Pendulum[{q0, p0}, {pts, dt}, double] generates a list of
      {q, p, E} triples.  The initial position and momentum are
      denoted by \"q0\" and \"p0\"; the total number of points to
      return is given by \"pts\"; \"dt\" is the timestep size.  If
      you want to return every point twice (in order to plot a
      phase portrait with -2Pi < q < 2Pi instead of -Pi < q < Pi)
      then \"double\" should be True; otherwise \"double\" should
      be False.\n
      If \"double\" is omitted, as in Pendulum[{q0,p0}, {pts, dt}],
      its default value is False."
```

Copyright © 1998 David M. Harrison. This is version 1.2, data (m/d/y) 10/17/98,