Software Packages in Physics
Final Exam
2nd semester 2004-2005
6-June-2005

- ***Marking: Students are required to solve 3 out of the 4 problems given below, with problems 1 and 4 being mandatory. Problems 1 and 4 were marked out of 12, whereas 2 and 3 were marked out of 10, and the resulting mark was converted to be out of 20.***

Name:

Problem 1: The Perihelion of Mercury

Introduction

It has been long known that the orbit of the planet Mercury exhibits an anomaly that could not be explained by Newton's gravitational law. Only when Einstein's general relativity came along could the anomaly be satisfactorily explained.

When solving Newton's equations for the system in polar coordinates, we end up with an equation of radial distance between Sun and planet $r(\theta)$ describing an ellipse:

$$r(\theta) = r_{\min} \frac{1+e}{1+e\cos\theta}$$

Here, $r_{\min}$ is the distance to the point of closest approach of the planet to the Sun, and is called the *perihelion*. e is called eccentricity of the orbit. Especially for Mercury, this ideal equation, however, does not conform with reality; rather, it is observed that $r_{\min}$ changes direction constantly in a phenomenon termed *precession of perihelion*.

By including corrections from general relativity, we arrive at the following approximate, but more realistic, form:

$$\tilde{r}(\theta) \approx \frac{A}{1+B\cos(\Omega\,\theta)}$$

where A, B, and Ω are constants.

In this regime, it is predicted that the precession per revolution $\Delta\theta$ is given by:

$$\Delta\theta \approx \frac{6\pi G M_{\odot}}{c^2(1-e^2)R}$$

And this prediction has been verified experimentally, and is considered one of the early successes of Einstein's theory. Here, R is the semimajor axis of the planet's orbit, and AU = 1.495985×10^{11} m is the astronomical unit, which is a measure of the mean distance between the Earth and the Sun. $c = 3\times10^{8}$m/s is the speed of light, $G = 6.673\times10^{-11}$ is Newton's gravitational constant, and finally, $M_{\odot} = 1.989\times10^{30}$ kg is the Sun's mass. For Mercury, $R \equiv R_{\mercury} = 0.3871$ AU, and eccentricity of its orbit is $e_{\mercury} = 0.206$.

■ Statement of the problem

1. Find the precession per revolution for Mercury's orbit, and for Earth's orbit, (for Earth, $e_{\earth} = 0.0170$, and $R_{\earth} \simeq 1$ AU). Comment on what you get

2. From the package **Graphics`**, use the function **PolarPlot[]** to plot $\tilde{r}(\theta)$; once use $\Omega < 1$, and once use $\Omega = 1$. What do you notice?

■ Your solution of problem 1:

We start with the precession per revolution for Mercury $\Delta\theta_{\mercury}$:

$$\Delta\theta \approx \frac{6\pi G M_{\odot}}{c^2(1-e^2)R}$$

```
Δθ☿ = 6 G M π / (a c^2 (1 - e^2)) //.
  {e → 0.206, c → 3 * 10^8, a → 0.3871 AU, AU → 1.495985 * 10^11, G → 6.673 * 10^-11, M → 1.989 * 10^30}
```

5.01298×10^{-7}

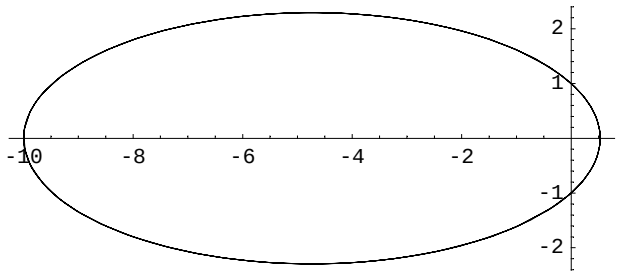
And now for Earth $\Delta\theta_{\earth}$:

```
Δθ⊕ = 6 G M π / (a c^2 (1 - e^2)) //.
  {e → 0.0170, c → 3 * 10^8, a → AU, AU → 1.495985 * 10^11, G → 6.673 * 10^-11, M → 1.989 * 10^30}
```

1.85872×10^{-7}

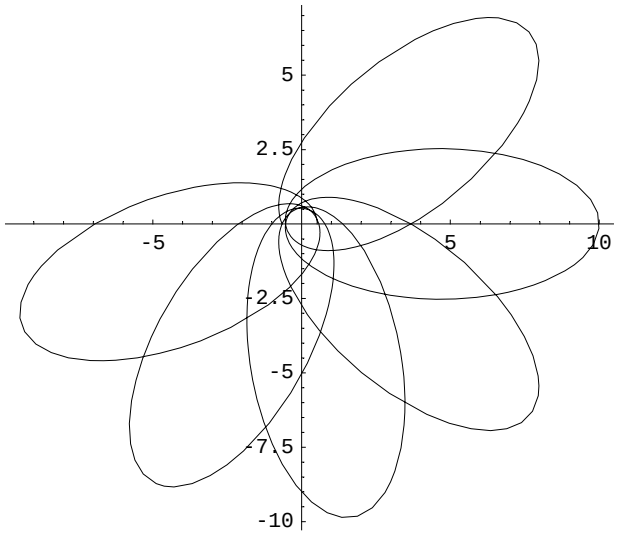
Plotting $\tilde{r}(\theta)$ for $\Omega = 1$:

```
<< Graphics`
r[θ_, Ω_] := 1/(1 + e Cos[Ω θ])
PolarPlot[r[θ, 1] /. e → 0.9, {θ, 0, 8 π}, AspectRatio → Automatic];
```



And now for $\Omega = 0.9$:

```
PolarPlot[r[θ, 0.9] /. e → 0.9, {θ, 0, 13 π}, AspectRatio → Automatic, PlotRange → All];
```



For $\Omega = 1$, we clearly see the orbit is closed ellipse, as predicted by Newton's theory, whereas for $\Omega = 0.9$, the orbit is not closed, and θ must go beyond 2π in order to execute one cycle, this is what we meant by precession of the perihelion.

Problem 2: Newton's Method

■ Statement of the problem

Write a *Mathematica* programme in the functional programming style that implements Newton's method to find the extremum of a single-variable function. Include an option that gives the user the choice of stopping iteration either by pre-fixing the number of iterations to be carried out, or by stopping when a tolerance supplied by the user is reached.

Useful functions: **`Nest[]`**, **`NestList[]`**, **`NestWhile[]`**, and **`NestWhileList[]`**.

■ Your solution of problem 2:

The following is not purely functional, but it will do:

```
NewtonMethod[f_, {x_, x0_}, {mtd_, val_}] :=
 Which[mtd === MaxIterations, Nest[(# - f'[#]/f''[#]) &, x0, val],
  mtd === Tolerance, NestWhile[(# - f'[#]/f''[#]) &, x0, (Abs[f'[#]] > val) &]]
```

Let us apply on the function we used in the lecture:

```
f[x_] := x Sin[x + 1]
```

Using the number of iterations as stopping criterion, we get:

```
NewtonMethod[f, {x, 4.}, {MaxIterations, 20}]
```

```
3.95976
```

while, stopping with tolerance:

```
NewtonMethod[f, {x, 4.}, {Tolerance, 0.001}]
```

```
3.95976
```

Problem 3: The Runge-Kutta Method

■ Statement of the problem

In the last lecture we introduced several methods for solving differential equations numerically. To solve examples on the methods developed, we built *Mathematica* functions for that purpose. Build a similar function for implementing the Runge-Kutta method, and apply it on the following equation:

$$\frac{dy}{dx} = -2x - 2y, \text{ for } y(0) = -1 \text{ and } x \in [0, 1]$$

Take your step size to be $h = 0.1$, and tabulate your results appropriately.

■ Your solution of problem 3:

The next function is based on a function we developed in the last lecture:

```
RungeKuttaNDSolve[rhs1_, rhs2_, y_, {x_, xi_, xf_}, h_] :=
 Module[{ynew, yold = rhs2, acm = {{xi, rhs2}}, n, xnew, xold = xi, k},
  Do[k[1] = rhs1 /. {x → xold, y → yold}; k[2] = rhs1 /. {x → xold + h / 2, y → yold + h * k[1] / 2};
   k[3] = rhs1 /. {x → xold + h / 2, y → yold + h * k[2] / 2};
   k[4] = rhs1 /. {x → xold + h, y → yold + h * k[3]};
   ynew = yold + h (k[1] / 6 + k[2] / 3 + k[3] / 3 + k[4] / 6); xnew = xold + h;
   AppendTo[acm, {xnew, ynew}]; xold = xnew; yold = ynew, {n, xi, xf - h, h}]; acm]
```

Let us apply it on the given equation:

```
RungeKuttaNDSolve[-2 x - 2 y, -1, y, {x, 0, 0.4}, 0.1]
```

```
{{0, -1}, {0.1, -0.8281}, {0.2, -0.705486}, {0.3, -0.623225}, {0.4, -0.574002}}
```

And now let us compare with **`NDSolve[]`**'s result:

```
sln = NDSolve[{-2 x - 2 y[x] == y'[x], y[0] == -1}, y, {x, 0, 0.4}];
Flatten[Table[{x, y[x]} /. ss, {x, 0, .4, .1}], 1]
```

```
{{0, -1.}, {0.1, -0.828096}, {0.2, -0.70548}, {0.3, -0.623217}, {0.4, -0.573993}}
```

Close enough!

Problem 4: The 1D Wave Equation

■ Statement of the problem

A string stretched between two points is plucked and left to oscillate. The string is governed by the one-dimensional wave equation:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

where $u(x, t)$ is the amplitude of oscillation, and $c \equiv 2$. Given the following initial conditions:

$$u(x, 0) = 0, \qquad u_t(x, 0) = 3 \sin(\tfrac{\pi x}{9}),$$

and the following boundary conditions:

$$u(-9, t) = 0, \quad u(9, t) = 0,$$

solve the differential equation to find $u(x, t)$ for $x \in [-9, 9]$, $t \in [0, 20]$. Then use an appropriate function from the package **`Graphics`Animation`** to animate the solution, and finally redo the plot using **`DensityPlot[]`**.

What is the physical significance of the boundary conditions in this problem?

■ Your solution of problem 4:

In the actual exam, the initial value $u_t(x, t)$ *was given as* $3\sin(\frac{\pi x}{9}) - 1$, *when in fact it should have been* $u_t(x, 0) = 3\sin(\frac{\pi x}{9})$. *Not using that would not lead to the desired result. Any student who reached the stage of writing the correct form of the* **NDSolve[]** *statement got the full mark even though subsequent steps could not be obtained as desired.*

```
sln = NDSolve[{D[u[t, x], t, t] == 2 D[u[t, x], x, x],
   u[0, x] == 0, Derivative[1, 0][u][0, x] == 3 Sin[π x / 9],
   u[t, -9] == u[t, 9] == 0}, u, {t, 0, 20}, {x, -9, 9}]
```

```
{{u → InterpolatingFunction[{{0., 20.}, {-9., 9.}}, <>]}}
```

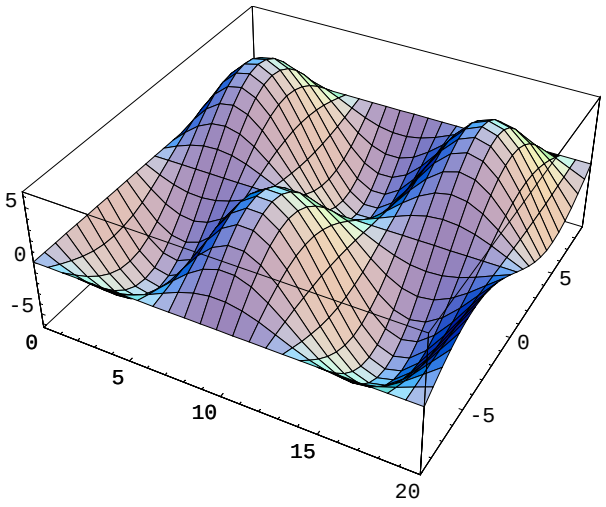
```
<< Graphics`Animation`
```

Here's a function from that package that does the required task, we omit the output due to its size. The output is a sequence of snap-shots of the solution, double-clicking any of these frame will animate the solution.

```
MoviePlot[u[t, x] /. sln, {x, -9, 9}, {t, 0, 20, 1}, AspectRatio → Automatic]
```

Instead of animation, we can make a 3D plot of *x*, *t*, and *u*:

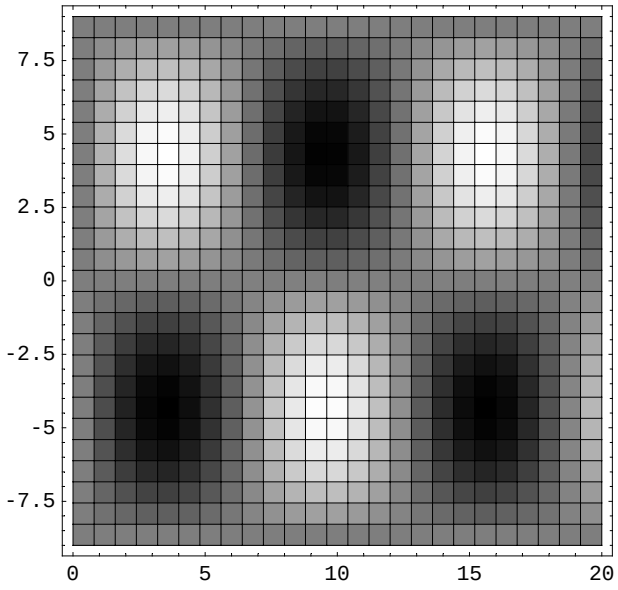
```
Plot3D[Evaluate[u[t, x] /. sln][[1]], {t, 0, 20}, {x, -9, 9}]
```




```
- SurfaceGraphics -
```

Or an appropriate representation of such a 3D plot, for instance:

```
DensityPlot[Evaluate[u[t, x] /. sln][[1]], {t, 0, 20}, {x, -9, 9}]
```




```
- DensityGraphics -
```

The graphs above clearly show that the string would oscillate in time. The boundary condition $u(-9, t) = 0$, $u(9, t) = 0$ means that the string is held fixed at both ends thoughout the motion.