PHYS 410/555: Computational Physics Fall 2004 Homework 5
DUE: Thursday, December 2, 10:00 AM Report bugs to choptuik@physics.ubc.ca

*The following assignment, should you decide to accept it, involves writing and testing two Fortran 77 programs (***including makefiles***) which use* `lsoda` *to solve ordinary differential equations. As usual, all files required by the assignment must reside in the correct places on your* `lnx` *account for the homework to be considered complete. Contact me immediately if you are having undue difficulties with any part of the homework.*

These instructions will NEVER self-destruct.

Problem 1: Consider the following ODE, known as the Van der Pol equation:

$$\frac{d^2u(t)}{dt^2} - \epsilon\left(1 - u(t)^2\right)\frac{du(t)}{dt} + u(t) = 0 \qquad t \ge 0 \tag{1}$$

where $\epsilon > 0$ is a specified *positive* constant. Physically, this ODE describes the voltage behaviour of a tunnel diode oscillator, although for the purposes of this homework we will simply be viewing it as providing an interesting example of non-linear oscillator dynamics.

In directory ~`/hw5/a1`, write a `Fortran 77` program `vdp.f`, with corresponding executable `vdp` which solves the above ODE subject to the initial conditions:

$$u(0) = \mathtt{u0} \tag{2}$$

$$\frac{du}{dt}(0) = \mathtt{du0} \tag{3}$$

`vdp` must have the following usage:

```
usage: vdp <tmax> <u0> <du0> <epsilon> <tol> <olevel>
```

where

1. `tmax` is the `real*8` final integration time (`tmax` > 0).
2. `u0` is the `real*8` initial oscillator displacement.
3. `du0` is the `real*8` initial oscillator velocity.
4. `epsilon` is the `real*8` value of ϵ (`epsilon` > 0).
5. `tol` is the `real*8` `lsoda` tolerance. Use `itol = 1`, and pure absolute tolerance. ($10^{-12} \le$ `tol` $\le 10^{-2}$)
6. `olevel` is the `integer` output level. (`olevel` > 0).

Your program must be commented, and must perform rudimentary error-checking of command-line arguments: arguments should be of the proper type, and should be in the appropriate ranges as given above (as usual, you can use `i4arg` and `r8arg` to detect whether an argument is of the correct type by using default values which a user is unlikely to enter).

At requested output times, `vdp` must write on standard output the time, and the computed displacement and velocity of the oscillator at that time, using an output statement such as

```
write(*,*) t, y(1), y(2)
```

The requested output times, t_j, are defined by

$$t_j = 0,\ h,\ 2h,\ \cdots\ \mathtt{tmax} \tag{4}$$

where h is given by

$$h \equiv \frac{\mathtt{tmax}}{2^{\mathtt{olevel}}} \tag{5}$$

To test your implementation, code an independent residual evaluation program, `chk-vdp.f`—with corresponding executable `chk-vdp`—which applies an $O(h^2)$ finite-difference approximation of (1) to the output of `vdp`. `chk-vdp.f` should be identical in construction, as well as what it inputs and outputs, to the `chk-tlsoda.f` example covered in class, except that `chk-vdp.f` is to accept a single, `real*8` command-line argument, `epsilon`. Use the usual $O(h^2)$ difference approximations for the first and second time-derivatives in (1); namely:

$$\frac{du}{dt}(t_j) = \frac{u(t_{j+1}) - u(t_{j-1})}{2h} + O(h^2) \tag{6}$$

$$\frac{d^2u}{dt^2}(t_j) = \frac{u(t_{j+1}) - 2u(t_j) + u(t_{j-1})}{h^2} + O(h^2) \tag{7}$$

Create a script `Do-chk` which contains the following

```
#!/bin/sh

epsilon=1.0
vdp 10.0 0.0 1.0 $epsilon 1.0d-8 10 | nth 1 2 > vdp-out

for inc in 8 4 2 1; do
   nth 1 2 < vdp-out | lines 1 . $inc | chk-vdp $epsilon
done
```

Once you are satisfied that both `vdp` and `chk-vdp` are working properly, execute the script and save the output as `chk-vdp-out`.

In order to get a feel for the behaviour of the oscillator, use `vdp` to investigate the solution of (1) for a variety of values of ϵ ($0 \le \epsilon \le 5$ suggested), `u0` ($10^{-3} \le$ `u0` ≤ 10 suggested) and `du0`, ($-5 \le$ `du0` ≤ 5 suggested). When performing your studies, ensure that you specify `tmax` large enough to determine the long-time behaviour of the oscillator. You can use `gnuplot` to plot the results of your computations.

An interesting way to examine the dynamics of an oscillator is to make a *phase-space plot*, i.e. a parametric plot (in t) of the oscillator's velocity versus its displacement. Use `gnuplot` to make a *single* phase-space Postscript plot called `vdp-phase` showing the du/dt vs u trajectories from the following three computations:

```
vdp 50.0   0.01  0.0  1.0 1.0e-8 11
vdp 50.0    1.0 -1.0  1.0 1.0e-8 11
vdp 50.0    3.0  1.0  1.0 1.0e-8 11
```

Using `gnuplot`, make a Postscript plot called `vdp.ps` showing the oscillator displacement versus t for the following computation

```
vdp 100.0 0.01 0.0 10.0 1.0e-8 12
```

What can you say about the long-time behaviour of the oscillator as a function of $u(0)$ and $du/dt(0)$ for fixed ϵ? Answer in `~/hw5/a1/README`.

Problem 2: Consider our old friend from the previous homework, the (non-dimensionalized) *diffusion equation*, or *heat equation*, for the function $u \equiv u(t, x)$:

$$u_t = u_{xx} \quad \text{on} \quad 0 \le x \le 1\,, \;\; 0 \le t \le t_{\max} \tag{8}$$

subject to (i) the initial conditions:

$$u(0, x) = u_0(x), \tag{9}$$

(where $u_0(x)$ is a specified function), and (ii) the boundary conditions:

$$u(t, 0) = u(t, 1) = 0 \tag{10}$$

Problem 2a)
In directory ~`/hw5/a2`, write a `Fortran 77` program, `diffusion-mol.f`, with corresponding executable `diffusion-mol`, which solves the above PDE using the method of lines, and `lsoda`. Specifically, convert (8) into a system of N_x coupled ODEs by (i) introducing a regular mesh x_j and discrete unknowns u_j:

$$x_j \equiv (j-1)h \quad j = 1, 2, \cdots N_x \qquad h \equiv (N_x - 1)^{-1} \qquad u_j(t) \equiv u(x_j, t) \tag{11}$$

(ii) replacing u_{xx} with the usual $O(h^2)$ finite difference approximation, yielding

$$\frac{d\,u_j(t)}{dt} = \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} \qquad j = 2, \cdots N_x - 1 \tag{12}$$

and (iii) incorporating the Dirichlet boundary conditions (10) as follows:

$$\frac{du_1(t)}{dt} = 0 \tag{13}$$

$$\frac{du_{N_x}(t)}{dt} = 0 \tag{14}$$

We will code `diffusion-mol` so that, as much as feasible, it has the same command line arguments (including initial data type, and output type) as the finite difference programs `diffusion2` and `diffusion4` (`diffusion[24]`) from the previous homework, and so that it produces analogous output.
In particular, `diffusion-mol` will call `lsoda` to compute approximate solution values, $u_j^n \approx u(t^n, x_j)$, where the discrete times, $t^n, n = 0, 1, \ldots N_t$ are defined by

$$t^n = n\lambda h,\ n = 0, 1, \ldots N_t\,. \tag{15}$$

Here, $\lambda \equiv \Delta t/\Delta x \equiv \Delta t/h$ is the Courant number, and the number of time steps, N_t, is defined implicitly via

$$N_t \lambda h \equiv t_{\max}. \tag{16}$$

We stress that we introduce λ here solely to keep the usage of `diffusion-mol` as close as possible to that of `diffusion[24]`, and that, as was the case for the finite difference programs, λh and $t_{\max}$ must be commensurate (i.e. we must have $\mathrm{mod}(t_{\max}, \lambda h) = 0$).

In addition, and again paralleling Homework 3, for the specific initial data choice

$$u_0(x) \equiv u(0, x) = \sin(2pix) \tag{17}$$

we can compute the exact solution, $u(t^n, x_j)$, the solution error e_j^n,

$$e_j^n \equiv u_j^n - u(t^n, x_j) \tag{18}$$

as well as the RMS (or ℓ_2-norm) of the solution error $e(t^n)$

$$e(t^n) \equiv \|e_j^n\|_2 \equiv \left(\frac{\sum_{j=1}^{j=N_x} \left(e_j^n\right)^2}{N_x} \right)^{1/2} \tag{19}$$

Then, as was the case for `diffusion[24]`, we will provide `diffusion-mol` with an option to compute and output the error in the `lsoda`-MOL solution u_j^n, when the exact solution is known.
Thus, `diffusion-mol` must have the following usage:

```
usage: diffusion-mol <level> <olevel> <ostride> <tmax> <lambda> <id type> <out type> <tol>
```

Note that *all 8* arguments are *required*, in contrast to `diffusion[24]`, where `<out type>` was *optional.*

The command line arguments are defined as follows:

1. `<level>` is the `integer` level of spatial discretization: $N_x = 2^{\texttt{<level>}} + 1$.

2. `<olevel>`: Combined with `<level>`, defines the frequency of output. Must satisfy `<olevel>`$\leq$ `<level>`. Output is performed every

$$\texttt{ofreq} = 2^{\texttt{level}-\texttt{olevel}}$$

time steps, starting from step 0 (the initial time, $t = 0$).

3. `<ostride>`: Positive integer ≥ 1, which controls amount of output performed at output times. Specifically, grid function values at every `<ostride>`-th spatial grid point are output, so if `<ostride> = 1`, all values are output, if `ostride = 2`, every second value is output etc. Useful for "thinning" output prior to plotting via `gnuplot` for large values of `<level>` (large N_x).

4. `<tmax>`: Maximum integration time, $t_{\rm max}$.

5. `<lambda>`: Courant factor, $\lambda \equiv \Delta t/\Delta x = \Delta t/h$

6. `<id type>`: Controls what type of initial data is used. Implemented values are precisely the same as for Homework 3, and are repeated here for convenience.
   - `<id type>` $= 0$: Initialize using $u(0, x) = \sin(2\pi x)$ (Solution is known in this case, e^n_j and $e(t^n)$ can be computed.)
   - `<id type>` $= 1$: Initialize using $u(0, x) = \exp\left(-\left(\left(x - 0.4\right)/0.07\right)^2\right)$ (Gaussian initial data. [1])

7. `<out type>`: Controls what type of output is performed *to standard output*. Implemented values are
   - `<out type>` $= 0$: Solution values $u^n_j, j = 1,$ `<ostride>` $+ 1, 2$ `<ostride>` $+ 1, \ldots, N_x$ are output in a form suitable for subsequent plotting using `gnuplot`'s *surface plotting* facility.
   - `<out type>` $= 1$: Provided that `<id type>` $= 0$, e^n_j and $e(t^n)$ are computed at output times and t^n and $e(t^n)$ are output, two numbers per line.

8. `<tol>` is the `real*8 lsoda` tolerance. Use `itol = 1`, and `atol = rtol = tol`. ($10^{-12} \leq$ `<tol>` $\leq 10^{-2}$)

Your program must be commented and must perform error-checking of command line arguments as in Problems 2 and 3 of the previous homework (be sure to incorporate all of the constraints on command line arguments that are given above).

In developing your program, you may find it useful to work from the main program for `diffusion2` (or `diffusion4`) from the previous homework. In particular, you are encouraged to use the `xvs` interface, as those codes do, so that you can use the `xvs` visualization server, as well as the `sdftoxvs` utility program, which sends `.sdf` files to `xvs`.

IMPORTANT: While developing and debugging your code, *do not* run with values of `<level>`larger than 9, or you are likely to encounter the error alluded to in the next subproblem, which is *optional* for 410 (the subproblem is optional, not the error!).

After your program has been debugged to your satisfaction (you will probably want to use `<id type>` $= 0$ for testing purposes), use it to generate a file `gaussian8` which contains the standard output of the invocation

```
diffusion-mol 8 6 4 0.25 0.5 1 0 1.0d-3
```

Using `gnuplot`, make a parametric surface plot of this data and save it as the Postscript file `gaussian8.ps`.

Problem 2b) (REQUIRED for 555 students, OPTIONAL for 410)

Copy the script `~/phys410/hw5/a2/Run` to your solution directory and invoke it. Note that this script assumes that you have a makefile in your solution directory, so be sure that you do.

[1] Note that strictly speaking, the gaussian initial data does *not* satisfy the boundary conditions, $u(t, 0) = u(t, 1) = 0$. However the small level of violation of the boundary conditions is negligible in comparison to the level of truncation (discretization) error inherent in the solution at the resolutions at which you will be running `diffusion2`.

What happens when you execute `Run`? Do you encounter an `lsoda` error? If so, what is the error return code, and what does it mean? (You may want to invoke `diffusion` "by hand" rather than in the context of the script, in order to see the full error messages produced by `lsoda`.) Provide answers to these questions in a `README` file in the solution directory.

If you *did* encounter an `lsoda` error when you executed `Run`, try to modify your code so that *all of the invocations* of `diffusion` performed by `Run` complete *with no error messages from* `lsoda`. (*Hint:* For large values of N_x, the system of ODEs resulting from the MOL discretization becomes stiff, so a Jacobian *will* be computed by `lsoda`. Consider changing the Jacobian type indicator (the argument `jt` to `lsoda`) and providing one or more optional inputs to the routine. Recall that the `integral` example in the ODE notes illustrates the use of optional inputs to `lsoda`.)

If and when you modify your code so that all invocations of `diffusion` executed by `Run` are successful, leave *all* postscript plots generated by the script in your solution directory, along with the surface plot `gaussian8.ps`. Finally, briefly discuss your interpretation of the various error plots produced by `Run` in your `README` file.