The following assignment involves writing and testing three Fortran 77 programs which use finite-difference techniques to solve various problems. Do all development and execution on `sgi1`. *As usual, all files required by the assignment must reside in the correct places on your* `sgi1` *account for the homework to be considered complete. Contact me immediately if you are having undue difficulties with any part of the homework.*

Problem 1: In directory `~/hw5/a1` on your `sgi1` account, create a source file `dpint.f` which defines a `real*8` function `dpint` having the following header:

```
real*8 function dpint(xto,x,f,n,rc)
   integer      n,      rc
   real*8       xto,    x(n),    f(n)
```

`dpint` is to return the value p(`xto`), where $p(x)$ is the polynomial of degree $n-1$ which interpolates the input (x, f) pairs (`x(i)`,`f(i)`), `i = 1 ... n`. `dpint` should use Neville's algorithm to evaluate p(`xto`) (see class notes and *Numerical Recipes*, Sec. 3.1). The routine should also provide error checking (all error messages should be directed to standard error), and set the return code, `rc`, as follows:

- If $n > 20$, print an error message stating that the requested degree of polynomial interpolation is too large, set `rc = 2` and return. Such a restriction is needed since the routine will require internal storage to implement Neville's algorithm.
- If the `x(i)` are not distinct, print a suitable error message, set `rc = 3`, and return.
- If `xto` $<$ $\min_i$ `x(i)` or `xto` $>$ $\max_i$ `x(i)`, then set `rc = 1` to indicate that *extrapolation* is occuring and compute the value of p(`xto`) (don't print an error message in this case).
- Normal interpolation, set `rc = 0`.

Working in the same directory, write a driver program called `tdpint` (source file `tdpint.f`, executable `tdpint`) which has the following usage:

```
tdpint: <xto> [<xto> ...]
```

`tdpint` must accept up to 10 `xto` values on the command-line, then read up to 20 (`x(i)`,`f(i)`) pairs from standard input. It is to then evaluate the interpolating polynomial (which passes through the (`x(i)`,`f(i)`) pairs) for each of the `xto` and output (`xto`,p(`xto`)) (two `real*8` numbers per line) on standard output. If a return code other than 0 or 1 is encountered, the main program should write an appropriate message to standard error, then exit. Note that you may wish to make use of the routines `dvvto, dvvfrom` you wrote for Homework 3. Test your implementations of `tdpint` and `dpint` thoroughly, both for valid and invalid input: I will test your work using my own input.

Problem 2: Consider the equation of motion for the displacement, $q(t)$, of a simple harmonic oscillator with frequency ω:

$$\ddot{q} = -\omega^2 q \tag{2.1}$$

where an overdot denotes differentiation with respect to time, t. Given initial conditions

$$q(0) = q_0 \qquad \dot{q}(0) = \dot{q}_0$$

the subsequent motion of the oscillator is completely determined via (2.1). In directory `~/hw5/a2` on your `sgi1` account, create a source file `sho.f` and corresponding executable `sho`, which solves this ordinary differential equation using a finite-difference technique. In particular, discretize time uniformly ($t^n = 0, \triangle t, 2\triangle t, 3\triangle t, \cdots$) and use the usual second-order ($O(\triangle t^2)$) approximation of the second derivative, $\ddot{q}$ to derive a discrete equation of motion. This equation of motion should be of the form:

$$q^{n+1} = c_0\, q^n + c_1\, q^{n-1}$$

for some coefficients c_0, c_1, where $q^n \equiv q(n\triangle t)$. Your program must accept command-line arguments (most of which will have defaults) as illustrated by the following usage message:

```
usage: sho <q0> [<qdot0> <omsq> <tmax> <level> <olevel>]

 defaults          0.0    1.0    8.0     8      8
```

The command line arguments have the following interpretation (data types shown in parentheses):

- `q0, qdot0`: Initial oscillator position, $q(0)$ and velocity, $\dot{q}(0)$, respectively (`real*8`).
- `omsq`: Square of oscillator frequency (i.e. ω^2) (`real*8`).
- `tmax`: Maximum (final) integration time (`real*8`).
- `level`: Discretization level (`integer`). The integration interval `(0 ... tmax)` will be divided into `nt` = $2^{\texttt{level}}$ `+ 1` time steps; thus $\triangle t = \texttt{tmax}/2^{\texttt{level}}$
- `olevel`: Output level (`integer`). Must not be greater than `level`. This parameter controls the frequency of output (time and position, as stipulated below) as follows:

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

  Let `it` label the time step, with `it = 0, 1, ... nt - 1`. Then output is generated whenever

  $$\texttt{mod(it,ofreq)} \quad \texttt{.eq.} \quad 0$$

  A key motivation for having this additional argument is to provide a mechanism to keep the specfic output times fixed (by keeping `olevel` fixed) as the resolution is increased (i.e. as `level` increases). Note that if `olevel .eq. level`, then output occurs every timestep; if `olevel .eq. level - 1`, output occurs every two timesteps, etc.

`sho` must periodically write the integration time, t^n, and computed oscillator position, q^n, to standard output (two numbers per line) as described above. It must also initialize q^0 and q^1 from the command-line values `q0`, `qdot0` to $O(\triangle t^3)$ (i.e. up to *and including* terms of $O(\triangle t^2)$) using the same Taylor series technique discussed in the handout *Notes on the 1D Wave Equation*.

Convergence-test your program by performing the following runs

```
sho 1.0 0.0 1.0 8.0  8 8 > out8
sho 1.0 0.0 1.0 8.0  9 8 > out9
sho 1.0 0.0 1.0 8.0 10 8 > out10
```

Note that since the output level is fixed at 8, each of the output files `out8`, `out9`, `out10` should contain output at the same set of 257 times. Let q_l denote the level l solution. Demonstrate that your solution appears to be second order accurate by using `gnuplot` to graph $q_8 - q_9$ and $4(q_9 - q_{10})$ on the same plot. Save a postscript version of your plot in a file called `ctest.ps`.

Examine the output from

```
sho 1.0 0.0 1.0 512 8 8
sho 1.0 0.0 1.0 513 8 8
```

What happens to the estimated solution at level 8 when `tmax` > 512? What is the value of $\omega\triangle t$ when `tmax` $= 512$? Can you come up with an explanation for the observed behaviour? (Answer these questions in `~/hw5/a2/README`.)

Problem 3: In directory ~/hw5/a3 on your sgi1 account, create a source file wave1d.f, and corresponding executable wave1d, which uses second-order finite-difference techniques (as discussed in class) to solve the following one-dimensional wave equation for $u(x,t)$:

$$u_{tt} = u_{xx} \qquad 0 \le x \le 1 \quad t \ge 0 \tag{3.1}$$

$$u(x,0) = l(x) + r(x) \qquad u_t(x,0) = l'(x) - r'(x) \qquad u(0,t) = u(1,t) = 0. \tag{3.2}$$

Here $l(x)$ and $r(x)$ are, respectively, the left-moving and right-moving components of the solution at $t = 0$ and $'$ denotes differentiation.

wave1d must accept 6 arguments as illustrated by the following usage message:

```
usage: wave1d <level> <dt/dx> <ncross> <a left-mover> <a right-mover> <olevel>
```

The arguments have the following interpretation:

- level: Discretization level (integer). The spatial mesh will have nx $= 2^{\texttt{level}} + 1$ points.
- dt/dx: "Courant number" (real*8). Ratio of temporal spacing $\triangle t$ to spatial mesh-size $\triangle x$.
- ncross: Final integration time in units of "crossing times" (integer). A crossing time is the amount of time it takes for a signal to propagate across the solution domain. Since the wave speed in (3.1) is 1, and the spatial domain is $0 \le x \le 1$, the crossing time in this case is also 1. Note that the number of time steps in the integration, nt, is then implicitly defined by level, dt/dx and ncross.
- <a left-mover>: The amplitude (real*8) of the initially left-moving component of the wave (see below).
- <a right-mover>: The amplitude (real*8) of the initially right-moving component of the wave (see below).
- olevel: Output level (integer). Must not be greater than level. This parameter defines the frequency of output as in the previous question:

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

  In addition, in this case ofreq also specifies a "spatial" frequency of output, e.g. if ofreq = 2, then at output times, every second value $u_1^n, u_3^n, \cdots$ is dumped (see below).

To aid in the development of your program, the following routines are provided in the file ~phys410/hw5/a3/util.f

The first routine is

```
subroutine dvgaussian(g,dg,ddg,x,n,amp,x0,del)
   integer        n
   real*8         g(n),     dg(n),     ddg(n),     x(n)
   real*8         amp,      x0,        del
```

which given x(j) , j = 1 ... n, amp, x0 and del returns a Gaussian profile, g(j), and its first two derivatives, dg(j) and ddg(j), evaluated on the finite-difference mesh:

$$\texttt{g(j)} \equiv g(\texttt{x(j)}) = \texttt{amp} \times \exp\left(-\left(\texttt{x(j)} - \texttt{x0}\right)^2 / \texttt{del}^2\right)$$

$$\texttt{dg(j)} \equiv \frac{dg}{dx}(\texttt{x(j)})$$

$$\texttt{ddg(j)} \equiv \frac{d^2g}{dx^2}(\texttt{x(j)})$$

This routine should be used to set up the initially left- and right-moving components of the solution, $l(x)$ and $r(x)$ as well as the first and second derivatives of these components, $l'(x), r'(x), l''(x), r''(x)$. Each of the two components should be a Gaussian with `x0 = 0.5d0`, `del = 0.1d0` and an amplitude given by the corresponding command-line argument. The difference values u_j^0 and u_j^1 are to be initialized to $O(\triangle t^3)$ using the Taylor series approach discussed in *Notes on the 1d Wave Equation*.

The second routine provided in `util.f` is

```
subroutine gnuout(u,x,nx,t,stride)
   implicit       none

   integer        nx,           stride
   real*8         u(nx),        x(nx),        t

   integer        j

   do j = 1 , nx , stride
      write(*,*) t, x(j), u(j)
   end do
   write(*,*)

   return

end
```

which writes data to standard output in `gnuplot` `splot`-format (see example source code, `gpwave.f`, in the on-line finite-difference notes). Assuming that the grid function `u` has been declared via

```
real*8 u(maxnx,2)
```

then the *only* output (to standard out) from your program should be generated every `ofreq` steps using a call like

```
call gnuout(u(1,np1),x,nx,t,ofreq)
```

When you are satisfied that your program is working, generate some sample output using:

```
wave1d 8 0.5 3 0.5 1.0 5 > out8
```

and then use `splot` in `gnuplot` to produce a surface plot of the results. Save a postscript version of your plot in the file `out8.ps`. Now try

```
wave1d 8 1.00025 3 0.5 1.0 5 > out8uns
```

and use `gnplot` to make a postscript file `out8uns.ps` containing a surface plot of the results. What appears to be happening to the solution in this case? (Answer this question in ~`/hw5/a3/README`.)