PHYS 555B: Computational Physics
Homework 2 (Version 5.0, homework now complete!)
Due: Thursday, April 5, 2007, 9:30 AM
(Report bugs etc. to choptuik@phas.ubc.ca)

Important: *This assignment requires that you write three* `f77` *programs which involve the solution of time-dependent PDEs using finite difference approximations (FDAs).*

In all cases you will ultimately be constructing programs that approximately solve initial-boundary value problems using prescribed solution domains, FDAs, initial conditions and boundary conditions. As you develop your codes to satisfy the specific *homework requirements, you are free to construct and use related codes that, for example, solve the given PDE using different initial conditions than those that are to be used for the hand-in* per se. *In other words, do not feel overconstrained by the specific details—including command-line arguments to programs—given below, in the development of your solutions. Rather, experiment as you wish, and as needed.*

That said, once you have finished with any such experimentation, please ensure that your homework does *include programs with the correct names, command line arguments etc., and which function in the requested manner, including output to appropriately named* `.sdf` *files in all cases. As usual, this requirement of attention to details on your part is partly to ease in grading of the homework, but also because programming to precise specifications is a useful skill!*

Note: *Recall that the graphics-based programs,* `xvs` *and* `DV`*, which can greatly expedite the analysis of finite difference solutions of PDEs in 1 space dimension and time (*`xvs`*), or 2 and 3 space dimensions and time (*`DV`*), are available for your use on the* `lnx` *machines, and that data from* `.sdf` *files can be sent to those programs using the* `sdftoxvs` *and* `sdftodv` *commands, respectively. See the documentation available via the Course Related Sofware web page for additional information, and feel free to contact me should you have any questions about the use of* `xvs`*,* `DV`*,* `sdftoxvs`*,* `sdftodv`*, generation of* `.sdf` *files etc.*

You should find `xvs` *useful in the solution of the first two problems, and* `DV` *similarly useful for the third.*

Problem 1:

Note: *This is a perfect example of "this hurts me more than it hurts you" exercise, so please do not be intimidated or put-off by the length of the problem specification, which includes discussion of the* `cvtestsdf` *utility (used to compute convergence factors as defined in class), and other things that are not specific to this particular problem. In short, the code,* `wave1d.f`*, that you must write to solve this problem need not be very long! Further, apart from* `wave1d.f` *itself, the only additional files required to complete the question are described in section* **1a)** *below.*

Consider the 1+1 wave equation for a scalar function, $u \equiv u(x,t)$, as discussed in class,

$$u_{tt} = u_{xx} \,. \tag{1}$$

In this exercise, you will solve the above PDE as an initial-boundary value problem on the domain $0 \le x \le 1$, $0 \le t \le t_{\max}$, subject to homogeneous Dirichlet boundary conditions,

$$u(0,t) = u(1,t) = 0 \,. \tag{2}$$

The issue of initial conditions will be deferred for the moment. The FDA you are to use to solve (1) is the "standard $O(h^2)$ approximation" discussed in class. That is, introduce a uniform finite difference mesh, given by

$$x_j = (j-1)\triangle x \equiv (j-1)h \quad , \quad j = 1 \ldots n_x \,, \tag{3}$$

$$t^n = n \triangle t \equiv n\lambda h \quad , \quad n = 0 \ldots n_t \,, \tag{4}$$

$$u^n_j \equiv u(x_j, t^n) \,. \tag{5}$$

where the number of spatial grid points, n_x, is given by $n_x = h^{-1} + 1$, and the number of time steps (minus 1), n_t, is given by

$$n_t = \frac{t_{\max}}{\triangle t}\,, \tag{6}$$

where it is assumed that $t_{\max}$ is precisely divisable by $\triangle t$.

The "standard" second order discretization of (1-2) is then given by

$$u_j^{n+1} = 2u_j^n - u_j^{n-1} + \lambda^2\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right), \qquad j = 2, 3, \cdots, n_x - 1\ ,\ n+1 = 2, \ldots n_t\,, \tag{7}$$

$$u_1^{n+1} = u_{n_x}^{n+1} = 0 \qquad n+1 = 0, \ldots n_t\,. \tag{8}$$

As was emphasized in class, in order to initialize the scheme, we need to specify u_j^0 and u_j^1, $2 \le j \le n_x - 1$, which is equivalent (in the limit $h \to 0$) to specifying $u(x,0)$ and $u_t(x,0)$. These in turn are the initial conditions nominally required for the solution of (1).

Your chief task in this problem is the investigation of the convergence behaviour of the finite difference approximation for two different initialization schemes, which differ in the accuracy to which the values of u_j^1 are specified.

To that end, assume that $u(x,0)$ and $u_t(x,0)$ are known, and then consider the following two initialization schemes (both based of Taylor series expansion):

1. Second-order initialization:

$$\begin{aligned} u_j^0 &= u(x_j, 0)\,, \\ u_j^1 &= u(x_j, 0) + \triangle t\, u_t(x_j, 0)\,. \end{aligned} \tag{9}$$

2. Third order initialization

$$\begin{aligned} u_j^0 &= u(x_j, 0)\,, \\ u_j^1 &= u(x_j, 0) + \triangle t\, u_t(x_j, 0) + \frac{1}{2}\triangle t^{\,2} u_{tt}(x_j, 0)\,. \end{aligned} \tag{10}$$

Note that in the first initialization, the advanced values u_j^1 are computed to second order accuracy—i.e. the leading order error term is $O(\triangle t^{\,2})$—whereas in the second initialization, the leading order error term in the u_j^1 is $O(\triangle t^{\,3})$.

In ~`hw2/a1` on your `lnx` account, create an executable `wave1d` (corresponding `f77` source in `wave1d.f`) with usage

```
usage: wave1d <tmax> <lev> <olev> <lambda> <initord>
```

where the types and meanings of the various command line arguments are as follows:

- `<tmax> (real*8)`: Final integration time, $t_{\max}$
- `<lev> (integer)`: Discretization level: $n_x = 2^{<\texttt{lev}>} + 1$, $\triangle x = h = 2^{-<\texttt{lev}>}$
- `<olev> (integer)`: Output level; controls frequency of output. Must be greater than 1 and less than or equal to `<lev>`. `.sdf` output is performed every $2^{<\texttt{lev}>-<\texttt{olev}>}$ time steps
- `<lambda> (real*8)`: Courant number, $\lambda \equiv \triangle t\,/\,\triangle x$
- `<initord> (integer)`: Initialization order, with valid values as follows:
  1. `<initord> .eq. 2`: Initialize u_j^1 using (9)
  2. `<initord> .eq. 3`: Initialize u_j^1 using (10)

As mentioned in the preamble, you are free to experiment with whatever initial data you please as you develop `wave1d`. However, the final version of `wave1d` *must* use the following instructor supplied functions to perform the initializations (9) and (10):

```
real*8 function u0(x)
    real*8   x

real*8 function ut0(x)
    real*8   x

real*8 function utt0(x)
    real*8   x
```

These three functions return the values $u(x,0)$, $u_t(x,0)$ and $u_{tt}(x,0)$, respectively, for a specific (yet suitably generic) choice of initial data. You should ensure that the value of `x` supplied to any of these routines is constrained by $0 \leq$ `x` ≤ 1 in all cases. The functions are defined in the `f77` source file, `~/wave1d/wave1d_util.f`, which should be copied into your solution directory, then compiled and linked with your own code in `wave1d.f` to produce the executable. As usual your directory should contain a `[Mm]akefile` to aid in the executable-generating process, clean up of executables and `.o` files, etc.

Additionally, for the purposes of testing (and in lieu of implementation of an independent residual evaluator), `wave1d_util.f` contains an additional function

```
real*8 function uexact(x,t)
   real*8    x, t
```

that, for the initial data defined via `u0` and `ut0` *only*, and for $0 \leq t \leq 0.25$, *will* return a good approximation to the *exact* solution, $u(x,t)$.

You will probably find it most convenient to use the Fortran callable routine, `xvs` (not to be confused with the related `xvs` visualization utility),

```
subroutine xvs(name,time,x,y,nx)
   character*(*)   name                ! dataset name
   real*8          time                ! dataset time (scalar)
   integer         nx                  ! length of dataset
   real*8          x(nx),   y(nx)      ! x, y values of dataset (vectors)
```

to generate the required `.sdf` output, which, for the final version of `wave1d`, is the single grid function u_j^n, at discrete times t^n defined by `<tmax>`, `<lambda>`, `<lev>` and `<olev>`. Observe that executables that invoke `xvs` must link to some additional libraries. Specifically,

```
-lsvs -lbbhutil -lsv
```

must be included in the linking command that creates `wave1d`. Note that on the `lnx` machines the environment variable `LIBXVS` should evaluate to the above library specification string, enabling you to use `$(LIBXVS)` in your `[Mm]akefile` to link against the appropriate archives.

For any invocation of `wave1d`, the name of the `.sdf` file generated should be of the form

```
u-<initord>-<lev>.sdf
```

Thus, the invocation

```
% wave1d 0.5 9 8 0.5 2
```

should generate

```
u-2-9.sdf
```

while

```
% wave1d 0.5 11 8 0.5 3
```

should generate

```
u-3-11.sdf
```

Files with such names can be easily produced using the utility `character` function, `itoc`, that is defined in the `libp410f` library, and that can be used as illustrated by the following code fragment:

```
      character*2   itoc

      real*8        x(10000), u(10000)
      real*8        t
      integer       lev, initord, nx
              .
              .
              .
      initord = 3
      lev = 9
      call xvs('u-'//itoc(initord)//'-'//itoc(lev),t,x,u,nx)
              .
              .
              .
```

The call to `xvs` in the above will result in the creation of `u-3-9.sdf`. Note how `itoc` is declared, i.e. as type `character*2`, so that, irrespective of how many digits there are in the integer supplied as an argument to `itoc`, a length-2 character string will be returned, right-padded with spaces as necessary. Also note that in generating a filename from its first argument, `xvs` discards any characters, including white space, that are not alphanumeric, underscore, or minus sign.

Be cautious in your handling of command line arguments, the derivation of quantities, such as n_t, that depend the value of these arguments, and the output of data via calls to `xvs` at discrete time intervals controlled by `<lev>` and `<olev>`. In particular, a series of invocations such as

```
% wave1d 1.0  8 8 0.5 3
% wave1d 1.0  9 8 0.5 3
% wave1d 1.0 10 8 0.5 3
```

should result in three `.sdf` files containing the same number of datasets, defined at the name set of discrete times, t^n, including the initial and final times, $t = 0.0$ and $t = 1.0$, respectively.

The `cvtestsdf` utility

Recall from our in-class discussion of the convergence of finite difference approximations of time-dependent PDEs, that given finite difference solutions generated from the same initial conditions, but using discretization scales in a $1 : 2 : 4$ ratio, we can define a convergence factor, $Q(t)$

$$Q(t^n) \equiv \frac{\|u^{4h} - u^{2h}\|_2(t^n)}{\|u^{2h} - u^h\|_2(t^n)} \tag{11}$$

where $\|\cdot\|$ is the discrete ℓ_2 norm, the subtractions between grid functions are to be understood to be made on the set of grid points common between the two discrete domains, and the 3 grid functions must all be defined at the same discrete set of times, t^n. In the limit $h \to 0$, the convergence factor, $Q(t)$, should asymptote to 2^p for a p-th order accurate scheme.

Given three `.sdf` files, each of which contains one of u^{4h}, u^{2h} or u^h, `cvtestsdf` computes $Q(t^n)$ as defined by (11) and then outputs t^n and $Q(t^n)$ to standard output, two numbers per line.

You will use this utility to complete the current problem, and will likely find it convenient in investigating the convergence of the codes that you will write in the other two parts of the homework.

You can see an example of how `cvtestsdf` works, and what sort of output it produces, via the files in ~`phys410/cvtestsdf` on the `lnx` machines. Here's a session trace which shows the "demo" being executed as `matt@lnx1.physics.ubc.ca`.

```
% mkdir /tmp/demo
% cd /tmp/demo
% cp -a ~phys410/cvtestsdf .
% cd cvtestsdf

# Generate 'cvtestsdf' usage message
% cvtestsdf
usage: cvtestsdf <file 1> <file 2> <file 3> [<dstem> <base level>]

       Computes and outputs to standard output the time, t, and
       the three-level convergence factor, Q(t), defined by

              ||u^4h - u^2h||_2 (t)
       Q(t) = --------------------
              ||u^2h - u^h||_2 (t)

       As h-> 0, Q(t) should asymptote to 2^p for a pth order accurate scheme.

       Specify .sdf files containing single grid function (u) from coarsest to
       finest resolution. Grid resolutions must be in the ratio 4:2:1.
       All files must contain grid functions defined at
       the same output times.

       Supply <dstem> for pointwise-difference output to
       2 RNPL-style .sdf files with names of the form

          <dstem><l><l+1>.sdf
          <dstem><l+1><l+2>.sdf

       where <l> defaults to 0, but will be set to <base level>
       if that argument is supplied.

# Run 'wave1d' at levels 8, 9, 10, and with 'intord=3' ...
% wave1d 0.5  8 8 0.5 3
% wave1d 0.5  9 8 0.5 3
% wave1d 0.5 10 8 0.5 3
% ls *sdf
u-3-10.sdf  u-3-8.sdf  u-3-9.sdf


# ... and compute resulting convergence factor using 'cvtestsdf'
% cvtestsdf u-3-8 u-3-9 u-3-10
0 0
0.00195312 3.92482
0.00390625 3.90415
0.00585938 3.88856
0.0078125 3.87815
```

```
        .
        .
        .
0.492188 3.94048
0.494141 3.95249
0.496094 3.96272
0.498047 3.9714
0.5 3.97876
```

Note that at $t^n = 0$, `cvtestsdf` computes $Q(t^n) = 0$. You will frequently see this behaviour, which should be viewed as anomalous: because a closed-form prescription of the initial data is used by `wave1d`, equation (11) for $Q(t)$ evaluates to $0/0$, which `cvtestsdf` arbitrarily replaces with 0.

Problem 1a) Once you are satisfied that your version of `wave1d` has been implemented correctly, according to the specifications above, copy the script `wave1d-script` from `~/phys410/cvtestsdf` to the solution directory for this question, and then execute it. If you *have* implemented `wave1d` correctly, this will run calculations for both second and third order initializations, and for discretization levels 8 through 12 respectively. In addition, `cvtestsdf` will be invoked to compute $Q(t^n)$ for levels $8, 9, 10$, levels $9, 10, 11$ and levels $10, 11, 12$, and for both orders of initialization. The output from the six invocations of `cvtestsdf` will be captured in files

`cvt-2-10-11-12` `cvt-2-8-9-10` `cvt-2-9-10-11` `cvt-3-10-11-12` `cvt-3-8-9-10` `cvt-3-9-10-11`

where the filename-encoding of the initialization order and the discretization levels used in the computation of $Q(t^n)$ should be self-explanatory.

Using the above 6 files make a *single* postscript file, `cvt23.ps`, which shows $Q(t^n)$ versus t^n from all 6 invocations of `cvtestsdf` on a single plot. Your plot should use a y-range (i.e. range of $Q(t^n)$) of 1.5 to 4.5. Briefly summarize what you conclude about the convergence of your code for the two different values of `intord` in a `README` file in the solution directory.

I will also test your implementation of `wave1d` using invocations other than those performed by `wave1d-script`.

Problem 2: *The 1-d Time-dependent Schrodinger equation*

Note: *This problem requires that you use* `f77`*'s built in facilities for handling complex quantities. See the code* `democomplex.f`*, available via the "Miscellaneous" section of the Course Related Sotware web page for a brief demonstration of the use of the* `complex*16` *data type in* `f77`*, including basic arithmetic operations, as well as some of the more important intrinsic (built-in) functions that are available to manipulate complex values.*

Consider the non-dimensionalized form of the 1-d time-dependent Schrödinger equation for the complex (wave) function, $\psi \equiv \psi(t,x)$, as discussed in class,

$$i\psi_t = -\psi_{xx} + V(x)\psi\,, \tag{12}$$

where $V(x)$ is a specified potential function.

In this exercise, you will solve the above PDE on the domain $0 \le x \le 1, \;\; 0 \le t \le t_{\max}$, subject to homogeneous Dirichlet boundary conditions,

$$\psi(0,t) = \psi(1,t) = 0\,, \tag{13}$$

corresponding to infinite potential barriers at $x = 0$ and $x = 1$.

Again, the issue of initial conditions will be deferred for the time being.

Equation (12) is to be solved using the $O(h^2)$ Crank-Nicholson approximation also discussed in class. Introduce the exact same finite difference mesh used in the previous problem:

$$\begin{aligned} x_j &= (j-1)\triangle x \equiv (j-1)h \quad , \quad j = 1 \ldots n_x\,, & (14)\\ t^n &= n\triangle t \equiv n\lambda h \quad , \quad n = 0 \ldots n_t\,, & (15)\\ \psi_j^n &\equiv \psi(x_j, t^n)\,. & (16) \end{aligned}$$

where n_x and n_t are defined as previously.

Define the standard $O(h^2)$ approximation to the second spatial derivative via the difference operator, D_{xx}:

$$D_{xx}\,\psi_j^n \equiv \frac{\psi_{j+1}^n - 2\psi_j^n + \psi_{j-1}^n}{\triangle x^{\,2}}\,. \tag{17}$$

The Crank-Nicholson differencing of (12-13) is then given by

$$\begin{aligned} i\frac{\psi_j^{n+1} - \psi_j^n}{\triangle t} &= -\frac{1}{2}D_{xx}\left(\psi_j^{n+1} + \psi_j^n\right) + \frac{1}{2}V_j\left(\psi_j^{n+1} + \psi_j^n\right) & (18)\\ &\qquad j = 2, 3, \cdots, n_x - 1\,, \quad n+1 = 1, \ldots n_t\,, \\ \psi_1^{n+1} &= \psi_{n_x}^{n+1} = 0 \qquad n+1 = 1, \ldots n_t\,. & (19) \end{aligned}$$

Note that (18-19) constitute a complex *tridiagonal* linear system for the n_x advanced time unknowns, ψ_j^{n+1}.

Wave function normalization and auxiliary quantities

Recall the physical interpretation of

$$|\psi(x,t)|^2\,dx \equiv \psi(x,t)\psi^*(x,t)\,dx\,, \tag{20}$$

as being the probability of finding the particle described by the Schrödinger equation (12) in the interval $(x, x+dx)$ at time t. Thus, $\psi(t,x)$ is to be normalized so that

$$\int_0^1 |\psi(x,t)|^2\,dx = 1\,. \tag{21}$$

Note that if (21) is satisfied at the initial time, $t = 0$, then evolution of ψ via (12) guarantees that the wave function remains normalized at future and past times. Of course, this a statement concerning the *continuum* solution, and can not necessarily be expected to hold precisely in the discrete case.

The specific types of initial data (and in one instance, the exact solution) that you are to implement, as described below, will generally produce unnormalized wave functions. For your convenience, the source file ∼phys410/util/sch1d/sch1d_util.f defines a routine, psi_normalize

```
      subroutine psi_normalize(psi,x,nx)
         implicit    none

         integer     nx
         complex*16  psi(nx)
         real*8      x(nx)
```

which will unit-normalize an arbitrary discrete wave function using an $O(h^2)$ approximation to (21). Specifically, the complex*16 grid function psi is both an input and output argument. On input, it should contain values corresponding to an arbitrary unnormalized wave function: on output, it will then contain values that are unit-normalized with respect to the aforementioned second order approximation to (21).

As is frequently the case in computational physics, the fundamental solution, $\psi(x,t)$, is not the most convenient quantity for interpretation of results, visualization etc. (although the discrete solution ψ_j^n is certainly the quantity on which one should focus until convergence of the implementation has been established). In the current case, "auxiliary" or "derived" quantities that are of specific interest include: $|\psi(x,t)|^2$, the cumulative probability function, $P(x,t)$, defined via

$$P(x,t) \equiv \int_0^x |\psi(\tilde{x},t)|^2 \, d\tilde{x} \,, \tag{22}$$

as well as the time expectation value, $\langle P(x,t) \rangle$ of $P(x,t)$:

$$\langle P(x,t) \rangle \equiv \frac{\int_0^t P(x,\tilde{t}) d\tilde{t}}{\int_0^t d\tilde{t}} \,. \tag{23}$$

Note that the spatial derivative of $\langle P(x,t) \rangle$ is the time expectation value of the probability distribution:

$$\frac{\partial}{\partial x} \langle P(x,t) \rangle = \left\langle |\psi(x,t)|^2 \right\rangle \,, \tag{24}$$

and is also of interest for the purposes of analysis of the numerical results.

As a further aid to the development of your programs to solve (18-19), the following routine (as well as supporting code) is also defined in the source file ∼phys410/util/sch1d/sch1d_util.f:

```
       subroutine psi_aux(psi,psire,psiim,psimodsq,pcum,pcumtexp,
     &                    x,t,dt,nx)
         implicit    none

         integer     nx
         complex*16  psi(nx)
         real*8      psire(nx), psiim(nx), psimodsq(nx), pcum(nx),
     &               pcumtexp(nx), x(nx)
         real*8      t, dt
```

In this routine, the grid function psi (ψ_j^n) is the primary input—from it, the routine computes auxiliary grid functions as follows:

$$\texttt{psire} \;\equiv\; \texttt{Re}\left(\psi_j^n\right) \,, \tag{25}$$

$$\texttt{psiim} \equiv \texttt{Im}\left(\psi_j^n\right), \tag{26}$$

$$\texttt{psimodsq} \equiv \left|\psi_j^n\right|^2, \tag{27}$$

$$\texttt{pcum} \equiv P_j^n, \tag{28}$$

$$\texttt{pcumtexp} \equiv \langle P\rangle_j^n. \tag{29}$$

Note that `x` is the vector of spatial coordinates, while `t` and `dt` are the corresponding integration time, t^n, and time step, $\triangle t$, respectively. Further note that `pcumtexp` is both an input and output argument. On input, and for any discrete time other than the initial one (i.e. for $t^n \neq 0$), `pcumtexp` should contain $\langle P\rangle_j^{n-1}$—on output `pcumtexp` will contain $\langle P\rangle_j^n$. When `psi_aux` is invoked at the initial time (i.e. with `t` $= 0$), `pcumtexp` will be initialized to P_j^0. Finally, as can be seen from inspection of the source code for `psi_aux`, the computations of P_j^n and $\langle P\rangle_j^n$ are performed using $O(h^2)$ approximations, consistent with the truncation error of the basic scheme (18-19).

To complete this problem, you are to write two separate codes, each of which solves (18-19) for a particular combination of initial data type and potential. Specifically, the two codes will treat the following cases

1. *Evolution of an eigenstate for vanishing potential (free particle in a box)*

   Here the potential is given by

   $$V(x) = 0 \qquad 0 < x < 1, \tag{30}$$

   $$V(x) = \infty \qquad x = 0,\ x = 1, \tag{31}$$

   and the (exact) eigenstates are given by

   $$\psi(x,t) = \exp\left(-i\left(m\pi\right)^2 t\right)\sin(m\pi x) \quad m = 1, 2, \cdots \tag{32}$$

   where m is the "principal quantum number". (Kudos, Robert Kehoe, 2007, for catching the $-i \to i$ sign error. Clearly the instructor didn't test his exact solution implementation except at the initial time!)

   Important note: As intimated previously, the initial data prescriptions (32) and (37) define *unnormalized* wave functions. Thus, each time you use one of these expressions (in a routine that computes the exact solution, e.g.), then you must normalize the function. Again, this can be done with a single call to the instructor-supplied routine, `psi_normalize`.

   For specfied m (which will be treated as an adjustable parameter), the initial data is to be given by (32) evaluated at $t = 0$, that is

   $$\psi(x,0) = \sin(m\pi x). \tag{33}$$

   (Note that the initial wave function is purely real.)

2. *Evolution of a gaussian wave-packet, possibly boosted, with a single square well, or square barrier potential*

   In this case the potential is given by three parameters, $x_{\min}$, $x_{\max}$ and V_0, as follows

   $$V(x) = V_0 \qquad x_{\min} \leq x \leq x_{\max}, \tag{34}$$

   $$V(x) = \infty \qquad x = 0,\ x = 1, \tag{35}$$

   $$V(x) = 0 \qquad \text{otherwise.} \tag{36}$$

   For V_0 positive (negative), this corresponds to a single square barrier (well) of height (depth) $|V_0|$ spanning the interval $x_{\min} \leq x \leq x_{\max}$. The initial data is a gaussian wave packet, possibly boosted (i.e. given some linear momentum), and is also a function of three parameters x_c, x_w (the center and width of the packet) and p (the momentum). Specifically

   $$\psi(x,0) = \exp\left(ipx\right)\exp\left(-\left(\left(x - x_c\right)/x_w\right)^2\right). \tag{37}$$

In the solution directory for this assignment, ~/hw2/a2, you are to create executables sch1d_eig and sch1d_square (with corresponding source files containing the main programs sch1d_eig.f and sch1d_square.f, respectively) that have usages as follows

1. sch1d_eig

   ```
   usage: sch1d_eig <tmax> <lev> <olev> <lambda> <m>
   ```

   where the types and meanings of the various command line arguments are as follows

   - <tmax> (real*8): Final integration time, $t_{\max}$
   - <lev> (integer): Discretization level: $n_x = 2^{<\texttt{lev}>} + 1,\ \triangle x = h = 2^{-<\texttt{lev}>}$
   - <olev> (integer): Output level; controls frequency of output. Must be greater than 1 and less than or equal to <lev>. .sdf output is performed every $2^{<\texttt{lev}>-<\texttt{olev}>}$ time steps
   - <lambda> (real*8): Courant number, $\lambda \equiv \triangle t \,/\, \triangle x$
   - <m> (integer): Principal quantum number. Must be positive definite.

2. sch1d_square

   ```
   usage: sch1d_square <tmax> <lev> <olev> <lambda> <xc> <xw> <p> <xmin> <xmax> <V0>
   ```

   where the types and meanings of the various command line arguments are as follows

   - <tmax> (real*8): As above.
   - <lev> (integer): As above.
   - <olev> (integer): As above.
   - <lambda> (real*8): As above.
   - <xc> (real*8): Center, x_c, of initial gaussian pulse.
   - <xw> (real*8): Width, x_w, of initial gaussian pulse.
   - <p> (real*8): Momentum, p, of initial pulse.
   - <xmin> (real*8): Spatial location, $x_{\min}$, of start of potential barrier/well.
   - <xmax> (real*8): Spatial location, $x_{\max}$, of end of potential barrier/well.
   - <V0> (real*8): Value, V_0, of potential in barrier/well.

The two programs that you construct may (should?) share considerable code, and must adhere to the following:

1. *Solution of the FDA*

   The Crank-Nicholson FDA for the Schrödinger equation is to be solved using the LAPACK routine ZGTSV, which solves a complex*16 tridiagonal linear system. The calling sequence for ZGTSV is precisely the same as for DGTSV, except that ZGTSV expects a complex*16 array everywhere that DGTSV expects a real*8 one:

   ```
         SUBROUTINE ZGTSV( N, NRHS, DL, D, DU, B, LDB, INFO )
   *
   *  -- LAPACK routine (version 3.0) --
   *     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
   *     Courant Institute, Argonne National Lab, and Rice University
   *     September 30, 1994
   *
   *     .. Scalar Arguments ..
         INTEGER            INFO, LDB, N, NRHS
   *     ..
   *     .. Array Arguments ..
         COMPLEX*16         B( LDB, * ), D( * ), DL( * ), DU( * )
   *     ..
   ```

2. *Scaled independent residual evaluation*

   Scaled independent residuals, r_j^{n+1}, are to be computed and output (see below) using the following *first order* (i.e. $O(h)$ truncation error) discretization of (12)

$$r_j^{n+1} \equiv 2^\ell \left| i\frac{\psi_j^{n+1} - \psi_j^n}{\triangle t} + D_{xx}\,\psi_j^{n+1} - V_j\psi_j^{n+1} \right|, \quad (38)$$

$$j = 2, 3, \cdots, n_x - 1\ ,\ n+1 = 2, \ldots n_t\ ,\ ,$$

$$r_1^{n+1} = 0\,, \quad (39)$$

$$r_{n_x}^{n+1} = 0\,. \quad (40)$$

   Note the prefactor of 2^ℓ in the definition of r_j^{n+1}, where ℓ is the discretization level (i.e. $\ell = \langle$`lev`$\rangle$) Since the independent discretization is $O(h)$ accurate, multiplication of the bare residual by 2^ℓ should produce a quantity that is roughly discretization-level-independent, in the limit $h \to 0$. Observe that the independent residual is undefined at the initial time, and thus should not be computed or output for $n = 0$.

3. *Incorporation of instructor-supplied utility routines*

   Make a copy of ∼`phys410/util/sch1d/sch1d_util.f` in your solution directory, and incorporate the code contained therein into your executables so that the routines `psi_normalize` and `psi_aux` can be used in your implementations, as described above.

4. *Program output:* `.sdf` *files*

   The following six *real-valued* grid functions should be output to `.sdf` files using calls to `xvs`, and at a frequency controlled by the command-line arguments `lev` and `olev`, as for the `wave1d` problem:

   (a) $\mathrm{Re}\left(\psi_j^n\right)$: Real part of wave function

   (b) $\mathrm{Im}\left(\psi_j^n\right)$: Imaginary part of wave function

   (c) $|\psi_j^n|^2$: Squared-modulus of wave function

   (d) P_j^n: Cumulative probability distribution

   (e) $\langle P\rangle_j^n$: Time expectation value of cumulative prob. distribution

   (f) r_j^n: Independent residual

5. *Names of* `.sdf` *files*

   The type of evolution, the discretization level, `<lev>`, and, in the case of `sch1d_eig`, the principal quantum number `<m>`, should be encoded in the `.sdf` filenames as follows:

   (a) *Program* `sch1d_eig`

   `.sdf` files should be named as follows

```
psire-eig-<m>-<lev>.sdf
psiim-eig-<m>-<lev>.sdf
psimodsq-eig-<m>-<lev>.sdf
pcum-eig-<m>-<lev>.sdf
pcumtexp-eig-<m>-<lev>.sdf
irmod-eig-<m>-<lev>.sdf
```

   corresponding to $\mathrm{Re}\left(\psi_j^n\right)$, $\mathrm{Im}\left(\psi_j^n\right)$, $|\psi_j^n|$, P_j^n, $\langle P\rangle_j^n$ and r_j^n, respectively. For example, the invocation

```
% sch1d_eig 0.1 10 6 0.05 3
```

should produce the files

```
psire-eig-3-10.sdf
psiim-eig-3-10.sdf
psimodsq-eig-3-10.sdf
pcum-eig-3-10.sdf
pcumtexp-eig-3-10.sdf
irmod-eig-3-10.sdf
```

(b) *Program* `sch1d_square`

`.sdf` files should be named as follows

```
psire-square-<lev>.sdf
psiim-square-<lev>.sdf
psimodsq-square-<lev>.sdf
pcum-square-<lev>.sdf
pcumtexp-square-<lev>.sdf
irmod-square-<lev>.sdf
```

corresponding to $\mathrm{Re}\left(\psi_j^n\right)$, $\mathrm{Im}\left(\psi_j^n\right)$, $|\psi_j^n|$,P_j^n, $\langle P\rangle_j^n$ and r_j^n, respectively. For example, the invocation

```
% sch1d_square 0.1 11 5 0.05 0.5 0.075 0.0 0.6 0.8 -1000
```

should produce the files

```
psire-square-10.sdf
psiim-square-10.sdf
psimodsq-square-10.sdf
pcum-square-10.sdf
pcumtexp-square-10.sdf
irmod-square-10.sdf
```

Note that the `character` function `itoc`, described in the first problem, can be used here as well to encode `<lev>` and `<m>` into the `.sdf` names.

Code testing

It is up to you to convergence test your codes using one or more of

1. `xvs`'s convergence test facility
2. `cvtestsdf`
3. independent residual study
4. the exact solution, where available

in order to verify that your implementations are correct.

Problem 2a)

Once you are satisfied that your implementation of `sch1d_square` is correct, perform 4 runs of the program with `<V0>` = -10000.0, -12000.0, -13000.0 and -15750.0, respectively, and with the values of all other command-line arguments fixed as follows: `<tmax>` = 0.05, `<lev>` = 12, `<olev>` = 8, `<lambda>` = 0.05, `<xc>` = 0.25, `<xw>` = 0.075, `<p>` = 5.0, `<xmin>` = 0.60 and `<xmax>` = 0.80.

Use the results from these runs to prepare 4 plots that graph $\left\langle|\psi(x,0.05)|^2\right\rangle$ versus x, for the 4 different values of `<V0>`; i.e. the plots are to display the time expectation value of the probability distribution at the final integration time, $t = 0.05$. Your plots should be saved in postscript files named `V0-10000.ps`, `V0-12000.ps`, `V0-13000.ps` and `V0-15750.ps`, respectively.

Provide some brief comments concerning what you observe in, and deduce from, these plots in a `README` file in the solution directory.

Course software usage hints:

1. You can use the `xvs` visualization utility to (a) compute

$$\left\langle |\psi(x,t)|^2 \right\rangle = \frac{\partial}{\partial x} \langle P(x,t) \rangle \tag{41}$$

   from $\langle P(x,t) \rangle$, via numerical differentiation, and, (b) then save $\left\langle |\psi(x,t)|^2 \right\rangle$ in an `.sdf` file.

2. You can use the `sdfdump` utility in conjunction with the instructor supplied script, `nth`, to generate ASCII data suitable for plotting with `gnuplot` or other plotting packages. Specifically, if the file `f.sdf`, storing the grid function, f_j^n, contains (for example), 15 datasets, then the pipeline

```
sdfdump -i 15 f.sdf | nth 2 3
```

   will produce on standard output

$$\begin{array}{cc} x_1 & f_1^{15} \\ x_2 & f_2^{15} \\ & \vdots \\ x_{n_x} & f_{n_x}^{15} \end{array}$$

Both `sdfdump` and `nth` are available on the `lnx` machines. Contact the instructor should you wish to install these utilities on your own system(s) and/or should you have any questions about their use.

Problem 3: *Solution of the 2D diffusion equation using the ADI FDA.*

Consider the non-dimensionalized 2+1 diffusion equation for the scalar function $u \equiv u(x, y, t)$ as discussed in class,

$$u_t = u_{xx} + u_{yy}\,, \tag{42}$$

with unit-square domain

$$0 \le x, y \le 1\,, \tag{43}$$

subject to the initial conditions

$$u(x, y, 0) = u_0(x, y)\,, \tag{44}$$

where $u_0(x, y)$ is a specified function, and boundary conditions

$$u(0, y, t) = u(1, y, t) = u(x, 0, t) = u(x, 1, t) = 0\,. \tag{45}$$

Introduce a standard, uniform finite difference mesh:

$$\begin{aligned} x_i &= (i-1)\triangle x \equiv (i-1)h \quad , \quad i = 1 \ldots n_x && (46)\\ y_j &= (j-1)\triangle y \equiv (j-1)h \quad , \quad j = 1 \ldots n_y && (47)\\ t^n &= n\triangle t \equiv n\lambda h \quad , \quad n = 0 \ldots n_t && (48)\\ u^n_{i,j} &\equiv u(x_i, y_j, t^n) && (49) \end{aligned}$$

Note that $n_x = n_y = h^{-1} + 1$, and that hereafter we will denote either of n_x or n_y by n_s.

Define the usual centred, $O(h^2)$ difference operators, D_{xx} and D_{yy}, that approximate the continuum operators, ∂_{xx} and ∂_{yy}, respectively:

$$\begin{aligned} D_{xx}u^n_{i,j} &\equiv \frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\triangle x^{\,2}}\,, && (50)\\ D_{yy}u^n_{i,j} &\equiv \frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\triangle y^{\,2}}\,. && (51) \end{aligned}$$

Define the Crank Nicholson approximation to (42-45) as follows:

Discretization of interior PDE

$$\frac{u^{n+1}_{i,j} - u^n_{i,j}}{\triangle t} = \frac{1}{2}\left(D_{xx} + D_{yy}\right)\left(u^{n+1}_{i,j} + u^n_{i,j}\right) \quad , \quad i, j = 2 \ldots n_s - 1 \quad , \quad n = 0, 1, ...n_t - 1\,. \tag{52}$$

Discretization of initial conditions

$$u^0_{i,j} = u_0(x_i, y_j, t^0) \equiv u_0(x_i, y_j) \quad , \quad i, j = 2 \ldots n_s - 1\,. \tag{53}$$

Discretization of boundary conditions

$$\begin{aligned} u^n_{1,j} &= u^n_{n_s,j} = 0 \quad , \quad j = 1 \ldots n_s\ ,\ n = 0 \ldots n_t\,, && (54)\\ u^n_{i,1} &= u^n_{i,n_s} = 0 \quad , \quad i = 1 \ldots n_s\ ,\ n = 0 \ldots n_t\,. && (55) \end{aligned}$$

Again, the specific form of the discrete initial and boundary conditions ensures (discrete) compatibility of the initial conditions with the boundary conditions, and the usual caveat concerning the need to ensure that effects due to small numerical inconsistencies are sub-truncation error, holds.

As discussed in class, the ADI approximation to (42), written in the form

$$\left(1 - \frac{1}{2}\triangle t\, D_{xx}\right)\left(1 - \frac{1}{2}\triangle t\, D_{yy}\right)u^{n+1}_{i,j} = \left(1 + \frac{1}{2}\triangle t\, D_{xx}\right)\left(1 + \frac{1}{2}\triangle t\, D_{yy}\right)u^n_{i,j}\,, \tag{56}$$

has the same order of truncation error (i.e. $O(h^2)$) as the Crank-Nicholson (CN) approximation (52), where here and hereafter the discrete initial and boundary conditions are unchanged from the CN approximation, as above.

Further, using an intermediary grid function, $u^{\star}_{i,j}$ the solution of (56) can be split into two stages, as follows:

$$\left(1-\frac{1}{2}\triangle t\, D_{xx}\right) u^{\star}_{i,j} = \left(1+\frac{1}{2}\triangle t\, D_{yy}\right) u^{n}_{i,j} \quad , \quad i,j=2\ldots n_s-1 \quad , \quad n=0\ldots n_t\,, \tag{57}$$

$$\left(1-\frac{1}{2}\triangle t\, D_{yy}\right) u^{n+1}_{i,j} = \left(1+\frac{1}{2}\triangle t\, D_{xx}\right) u^{\star}_{i,j} \quad , \quad i,j=2\ldots n_s-1 \quad , \quad n=0\ldots n_t\,. \tag{58}$$

Equations (57-58) together with the initial conditions (53) and boundary conditions (54-55) define the complete ADI FDA.

In the solution directory for this assignment, create the subdirectory `a3`, which is to contain an implementation of the ADI equations for the 2D diffusion equation, as described above. Specifically, you are to create an executable, `diff2dadi`, with one or more corresponding `f77` files, including `diff2dadi.f`, which should contain the main program. `diff2dadi` has usage

```
usage: diff2dadi <tmax> <lev> <olev> <lambda>
```

where the types and meanings of the various command-line arguments are as follows:

- `<tmax> (real*8)`: Final integration time, $t_{\max}$
- `<lev> (integer)`: Discretization level: $n_x = n_y = n_s = 2^{\texttt{<lev>}} + 1,\ \triangle x = \triangle y = h = 2^{-\texttt{<lev>}}$
- `<olev> (integer)`: Output level; controls frequency of output. Must be greater than 1 and less than or equal to `<lev>`. `.sdf` output (see below) is performed every $2^{\texttt{<lev>}-\texttt{<olev>}}$ time steps
- `<lambda> (real*8)`: Courant number, $\lambda \equiv \triangle t\,/\,\triangle x$

Your program should adhere to the following:

- *Solution of the FDA*

  Implement the ADI equations precisely as defined above, and use the `LAPACK` tridiagonal solver `DGTSV` to solve all tridiagonal systems that arise in the ADI solution.

- *Program output:* `.sdf` *files*

  Periodically, with a frequency controlled by the command-line arguments `<lev>` and `<olev>`, as in previous questions, your program should use one of the following routines to output the computed solution to an `.sdf` file named `u-<lev>.sdf`, where `<lev>` is to replaced with the actual integer value of `<lev>` that is supplied on the command line:

```
subroutine gft_out_bbox(name,time,shape,rank,bbox,data)
   character*(*)    name     ! dataset name
   real*8           time     ! dataset time
   integer          shape(2) ! shape of data: [nx ny]
   integer          rank     ! dataset rank (number of dimensions), 2 in this instance
   real*8           bbox(4)  ! bounding box for data: [xmin xmax ymin ymax]
   real*8           data(shape(1),shape(2))  ! 2D dataset

subroutine gft_out_brief(name,time,shape,rank,data)
   character*(*)    name     ! dataset name
   real*8           time     ! dataset time
   integer          shape(2) ! shape of data: [nx ny]
   integer          rank     ! dataset rank (number of dimensions), 2 in this instance
   real*8           data(shape(1),shape(2))  ! 2D dataset
```

If you use the `gft_out_brief` routine, you should call `gft_out_set_bbox`

```
subroutine gft_out_set_bbox(bbox,rank)
   integer          rank     ! rank of datasets to be output
   real*8           bbox(*)  ! bounding box for data: [xmin xmax ymin ymax]
```

before any calls to `gft_out_brief`, in order to set the bounding box to [0.0 1.0 0.0 1.0]. (The default for 2D data is to assume a domain defined by the bounding box [-1.0 1.0 -1.0 1.0].)

As usual, let the instructor know immediately should you have difficulty with these routines, or if you have questions about their use and/or visualization of the contents of 2D `.sdf` files with the `DV` application that was demoed over mid-term break.

- *Initial data*

  As you develop your program, you are free to use initial data of your own choosing. However, the final version of your code should "hard-code" the following specific initial conditions

$$u(x,y,0) = u_0(x,y) = \exp\left(-\left(\left(\frac{x-0.6}{0.05}\right)^2 + \left(\frac{y-0.7}{0.10}\right)^2\right)\right) . \tag{59}$$

  Note, however, that the above expression is to be used only for the *interior* grid points at $t = 0$. The boundary values at the initial time should be set to 0.0 per the discrete boundary conditions (54-55).

- *Independent residual evaluation and output*

  In this instance there is, of course, a very natural independent residual with which one can test the implementation of the ADI scheme—namely that provided by the Crank-Nicholson discretization (52) itself. Specifically, momentarily denoting the solution of the ADI FDA by $\tilde{u}^n_j$, then the Crank-Nicholson residual, $r^{n+1/2}_{i,j}$, is given by

$$r^{n+1/2}_{i,j} \equiv \frac{\tilde{u}^{n+1}_{i,j} - \tilde{u}^{n}_{i,j}}{\triangle t} - \left[\frac{1}{2}\left(D_{xx} + D_{yy}\right)\left(\tilde{u}^{n+1}_{i,j} + \tilde{u}^{n}_{i,j}\right)\right] . \tag{60}$$

  As discussed in class (in the general context of independent residual evaluation), viewing r as a function defined on the continuum, we can expect the following asymptotic behaviour:

$$\lim_{h\to 0} r(t,x;h) = h^2 r_2(t,x) + \text{higher order}\,, \tag{61}$$

  where r_2 is some h-independent function.

  Important note: Observe that the residual (60) is naturally defined at discrete times $t^{n+1/2} \equiv t^n + \triangle t\,/2, n = 0, \ldots n_t - 1$. From the point of view of the usual sort of convergence tests discussed in the course, one consequence of this fact is that it will be generally impossible to *directly* compare the residuals computed at different levels, ℓ, of discretization, with corresponding mesh spacings $h_\ell = h_{\ell-1}/2$. That is, although the discrete times, $t^n_{\ell-1}$, on a coarse grid will be a subset of the fine grid times, t^n_ℓ, *none* of the values $t^{n+1/2}_{\ell-1}$, from the coarse grid will correspond to the "half time step" values, $t^{n+1/2}_\ell$ on the fine grid. Thus, a direct comparison of independent residuals computed on different levels would require interpolation in time of the residuals, and would constitute an overly cumbersome procedure, given the purpose of the residual evaluation.

  Given this observation, consider computing the spatial ℓ_2 norm (RMS value), $\|r\|_2$, of r at each discrete time step. Clearly, $\|r\|_2$ should also be an $O(h^2)$ quantity, so that if we compute a "level-scaled" residual norm, $\hat{r}(t;h)$, defined by

$$\hat{r}\left(t^{n+1/2};h_\ell\right) \equiv 4^\ell \left\| r^{n+1/2}_{i,j} \right\|_2 , \tag{62}$$

then we should find that $\hat{r}(t,h)$ is an approximately h-independent function. Further, it is extremely improbable that $\|r\|$ can be $O(h^2)$ without (61) being satisfied, so that we can essentially view the constancy of $\hat{r}$, as $h \to 0$, as a *sufficient* as well as *necessary* condition for the implementation of the ADI FDA to be correct.

Thus, for time steps, $n = 0 \ldots n_t - 1$, your program should compute the scaled residual norm (62), and output the base-10 logarithm of that quantity, along with the corresponding discrete time, $t^{n+1/2}$, to *standard output*. Specifically, assuming that the `real*8` variable `scnrm2rhat` contains the scaled ℓ_2 norm of $\hat{r}$ defined by (62)—where the discretization level, ℓ, is to be identified with the command-line argument, `<lev>`—and that the `real*8` variables `t` and `dt` contain the discrete time, t^n, and time step, $\triangle t$, respectively, your program should include an output statement such as

```
write(*,*) t + 0.5d0 * dt, log10(scnrm2rhat)
```

Note that the standard output from invocations of `diff2dadi` with varying discretization levels can then be "overlaid" for comparison, with no further post-processing required, using, e.g., `gnuplot` or `xvs`.

In order to facilitate computation of the scaled independent residuals, your implementation of `diff2dadi` should make use of the instructor supplied routines for this problem that are defined in the source file `~phys410/util/diff2dadi/diff2dadi_util.f`. Specfically, you should copy this source file into your solution directory for this problem, and incorporate it into your executable. You should then ensure that the routine `clcresidcn` is called at each discrete time, $t^n, n = 0 \ldots n_t - 1$. Note that `clcresidcn` has the header

```
subroutine clcresidcn(r,unp1,un,nx,ny,dx,dy,dt)
   implicit    none

   integer     nx, ny
   real*8      r(nx,ny), unp1(nx,ny), un(nx,ny)
   real*8      dx, dy, dt
```

so that, in particular, storage must be passed in for an entire grid function's worth of residuals, `r(nx,ny)`. Given the array inputs `unp1` and `un`—which contain $u_{i,j}^{n+1}$ and $u_{i,j}^{n}$, respectively—as well as the self-explanatory scalar inputs, `nx`, `ny`, `dx`, `dy` and `dt`, `clcresidcn` will compute the *unscaled* CN residuals as defined by (60), and return the values in the output array, `r`. Following the call to `clcresidcn`, the ℓ_2 norm of the residuals can be computed using the `real*8` function `dmnrm2`

```
real*8 function dmnrm2(a,m,n)
   implicit       none
   integer        m,        n
   real*8         a(m,n)
```

which is also defined in `diff2dadi_util.f`.

As in the previous problem, it is up to you to ensure that your implementation of `diff2dadi` is correct (i.e. convergent) using independent residual evaluation as described above, as well as the `cvtestsdf` utility, and, if necessary, other testing procedures of your own design.

Problem 3a) Once you are confident that your implementation of `diff2dai` is correct, execute the following sequence of commands in your solution directory

```
% diff2dadi 0.0625 6 6 0.05 > cn-res-6
% diff2dadi 0.0625 7 6 0.05 > cn-res-7
% diff2dadi 0.0625 8 6 0.05 > cn-res-8
% diff2dadi 0.0625 9 6 0.05 > cn-res-9
```

Note that this should generate `.sdf` files `u_6.sdf`, `u_7.sdf`, `u_8.sdf` and `u_9.sdf`, and should capture the standard output (discrete time and $\log_{10}$ of the scaled independent residuals) of the invocations in the ASCII files `cn-res-[6789]`. Use `gnuplot` or another plotting package to make a postscript file named `cnres6789.ps` that shows the data from `cn-res-[6789]` on a *single* plot. **Important:** Ensure that you have used initial data of the form (59) for your final calculations and plot.