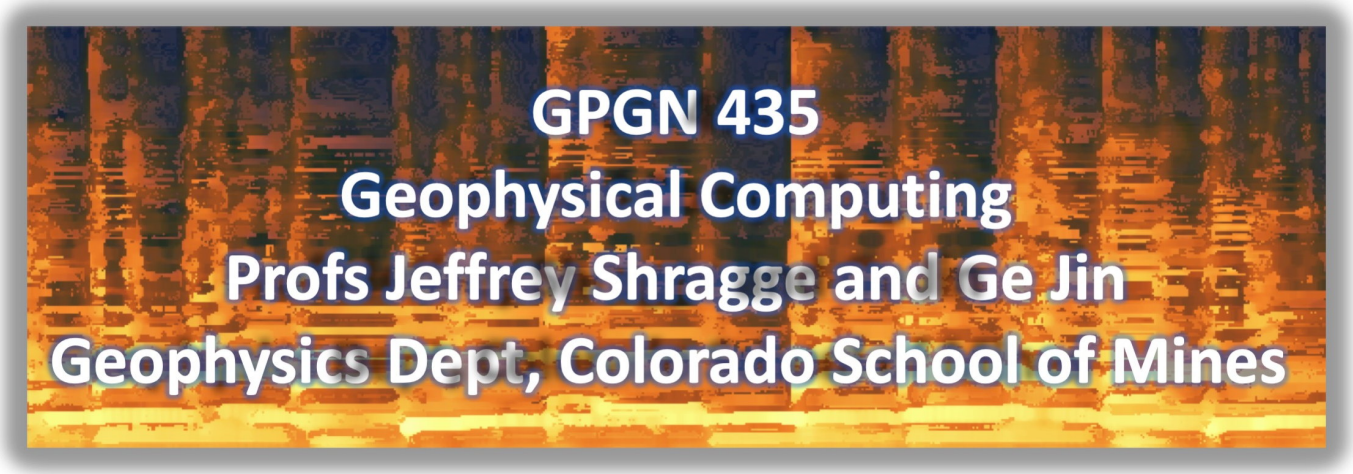
Out[1]: The raw code for this Jupyter notebook is by default hidden for easier reading. To toggle on/off the raw code, click here.



Module 9: Numerical Solutions to Parabolic PDEs

This module focuses on generating numerical solutions of **parabolic** partial differential equations (PDEs). Recall that these types of second-order PDEs

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu + G = 0 \tag{1}$$

are defined by the condition $B^2 - AC = 0$ (where y is considered the t variable and thus $A = E = 1$ and all others are equal 0).

Three interesting examples (with increasing complexity) are the following:

- **Heat Flow / Diffusion (Homogeneous medium)**: Used to model time-varying heat flow (or diffusive) phenomena with the rate of evolution is determined by the thermal conductivity κ (or diffusivity). The solution $\phi = \phi(x, y, z, t)$ is known on the boundary of the solution domain D (e.g., $\phi|_{\partial D} = F(x, y, z)$) as is the initial distribution $\phi_0 = \phi_0(x, y, z, t = 0)$:
$$\frac{\partial \phi}{\partial t} = \kappa \nabla^2 \phi. \tag{3a}$$
- **Heat Flow / Diffusion (Heterogeneous medium)**: This is like the example above, except now that the thermal conductivity (or diffusivity) is spatially varying, $\kappa = \kappa(x, y, z)$:
$$\frac{\partial \phi}{\partial t} = \nabla \cdot (\kappa \nabla \phi)\,. \tag{3b}$$
- **Convection - Diffusion**: These types of problems involve the types of heat flow (or diffusion) described above along with material transport at velocity $\mathbf{v} = [v_x, v_y, v_z]$ like we investigated in Module 7:
$$\frac{\partial \phi}{\partial t} = \nabla \cdot (\kappa \nabla \phi) + \mathbf{v} \cdot \nabla \phi. \tag{3c}$$

In the section below, we will look at generating solutions ϕ of the 1D and 2D heat flow / diffusion equation and then the 2D convection-diffusion equation throughout a computational domain D. We will also examine different boundary conditions (BCs) on boundary of D (defined by shortform $\delta\Omega$), namely the Dirchelet BC

$$\phi(\mathbf{x}) = f(\mathbf{x}), \quad \forall \mathbf{x} \in \delta\Omega$$

and the Neumann BC

$$\frac{\partial \phi}{\partial \mathbf{n}}(\mathbf{x}) = f(\mathbf{x}) \quad \forall \mathbf{x} \in \partial\Omega$$

where $\mathbf{n}$ is a normal vector defined at the grid boundary that points inward.

1D Heat Flow

Let's start with one of the more straightforward examples: 1D heat flow in a thin uniform metal bar with constant thermal conductivity κ. Let's say that the bar is of length L in the x direction, and is much thinner than L in both of the y and z dimensions. In this case, we can rewrite equation 3a with the following explict **partial differential equation (PDE)**:

$$\frac{\partial \phi}{\partial t} = \kappa \frac{\partial^2 \phi}{\partial x^2}. \tag{4}$$

For simplicity, let's say that the ends of the bar are fixed at some $\phi(x = 0, t) = A^\circ$C and $\phi(x = L, t) = B^\circ$C, which represent homogeneous Dirichelet **boundary conditions**. Finally, let there be an **initial condition** where the temperature given by $\phi(x, t = 0) = \phi_0(x)$. We can also say that the bar is perfectly thermally insulated along its length, and thus can only lose or gain heat at its ends. Thus, we have sufficient information to solve this heat flow problem.

Numerical solution domain

To compute a numerical solution, let's first **discretize** the solution space. In this case, we will segment the $x \in [0, L]$ into $nx + 1$ discrete element spaced $dx = L/nx$ apart. We will also break our time axis $t \in [0, t_{max}]$ into $nt + 1$ steps where $dt = t_{max}/nt$. This will give us the following example solution grid:

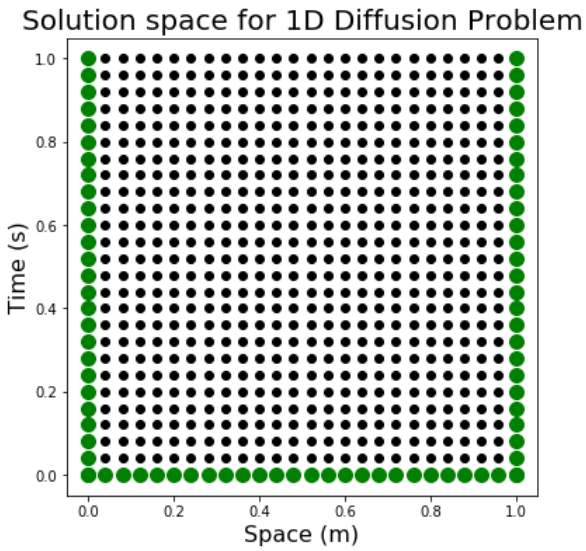


Figure 9-1. Example solution space for 1D diffusion problem. The green points are the locations where the solution is fixed by the initial condition (bottom) or the boundary conditions (left and right sides). The black dots are the locations where we need to compute the solution.

Explicit FTCS solution

The first numerical solution that we will investigate is an explicit method: forward in time, centered in space (FTCS). This method uses an $\mathcal{O}(\Delta t)$ forward approximation for the temporal first derivative and an $\mathcal{O}(\Delta x^2)$ centered approximation for the spatial second derivative. Thus, this numerical approximation can be written as:

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} = \kappa \frac{\left(\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n\right)}{\Delta x^2}. \tag{5}$$

Let's now multiply both sides by Δt and then rearrange equation 5 so that we have all terms at the $n + 1$ time step on the left-hand side, and all others on the right-hand side:

$$\phi_i^{n+1} = \alpha\phi_{i+1}^n + (1 - 2\alpha)\phi_i^n + \alpha\phi_{i-1}^n. \tag{6}$$

where $\alpha = \frac{\kappa \Delta t}{\Delta x^2}$ is introduced as a dimensionless number called the **diffusion number**.

Let's now code up a solver for this system:

Let's also define the *diffusion_1D_animate(i)* animate function once so that we can call it frequently below:

Copper and Iron Bars with Fixed Boundaries

Let's now create an example that calls our diffusion solver. We're going to discretize a solution domain $x \in [0, 1]$m with $nx = 51$ points ($dx = 0.01$m) and $t \in [0, 1]$s also with $nt = 101$ points ($dt = 0.01$s). As the initial distribution, let's use a Gaussian function

$$\phi_0(x) = 100\mathrm{e}^{-(x-x_0)^2/(2\sigma^2)} \tag{7}$$

where $x_0 = 0.5$m, $\sigma = 0.05$m and the units of ϕ are in °C. For the thermal diffusivity, we are going to look at two materials:

- Copper: $\kappa = 1.1 \times 10^{-4}$ m^2/2 (at 25°C)
- Iron : $\kappa = 2.3 \times 10^{-5}$ m^2/2

You'll note that the κ values are defined at a constant temperature. Generally speaking, the thermal diffusivity of materials can depend on the temperature itself, which will make the problem **non-linear** and more difficult to solve.

In this scenario, you might recall that the steady-state solution would be for the bars to become uniformly 0°C. Let's see if our numerical solution accomplishes that:

```
alpha (Copper) = 0.41250000000000003
alpha (Iron  ) = 0.08625
```

Out[5]:

Figure 9-2. Solutions of the 1D heat diffusion equation for an insulated 1m metal bar with the exposed ends held at 0ºC (i.e., zero Dirchelet boundary conditions). Solutions for copper (green) and iron (blue) are presented.

Playing with Boundary Conditions - Non-zero Dirichelet

Let's now look at the same example as above, but let's change the boundary conditions such that $\phi(x=0,t) = 0^{\circ}$C and $\phi(x=1,t) = 50^{\circ}$C. Let's rerun the code from above:

```
alpha (Copper) = 0.41250000000000003
alpha (Iron  ) = 0.08625
```

Out[6]:

Figure 9-3. Solutions of the 1D heat diffusion equation for an insulated 1m metal bar with the exposed ends held at 0ºC on the left and 50ºC on the right. (i.e., Dirchelet boundary conditions). Solutions for copper (green) and iron (blue) are presented.

Playing with boundary conditions - Neumann boundary conditions

The examples above looked at Dirichelet boundary conditions, which is where the boundaries are fixed at a certain value. However, there are important scenarios where you might want to control the heat gradient (i.e., $\frac{\partial \phi}{\partial x}$) at the boundary.

Let's say that we want to modify the above such that we had:

$$\frac{\partial \phi(x=0,t)}{\partial x} = A \tag{8}$$

We can think about applying this by discretizing equation 8 as:

$$\frac{U_1^n - U_0^n}{\Delta x} = A \tag{9a}$$

or solving for U_0^n:

$$U_0^n = U_1^n - A\Delta x. \tag{9b}$$

Similarly, at the right boundary we have

$$\frac{\partial \phi(x=L,t)}{\partial x} = B \tag{10}$$

and thus numerically we have:

$$\frac{U_N^n - U_{N-1}^n}{\Delta x} = B \tag{11a}$$

or solving for U_N^n:

$$U_N^n = U_{N-1}^n + B\Delta x. \tag{11b}$$

where the difference in sign in equations 9b and 11b is due to the opposing directions of normal derivative into the bar. Let's redo our solver above but use Neumann BC's:

Let's now rerun the example above but with Neumann boundary conditions of $\frac{\partial \phi(x=0,t)}{\partial x} = 1/\Delta x$°C and $\frac{\partial \phi(x=1,t)}{\partial x} = 0$°C

```
alpha (Copper) = 0.41250000000000003
alpha (Iron  ) = 0.08625
```

Out[9]:

0:00 / 0:07

Figure 9-3. Solutions of the 1D heat diffusion equation for an insulated 1m metal bar with the exposed ends experiencing a constant inward flux of (1/Δx)ºC/m on the left and 0ºC/m on the right. (i.e., Dirchelet boundary conditions). Solutions for copper (green) and iron (blue) are presented.

Stability of the FTCS method

One important consideration that have not discussed is whether or not the FTCS method is **unconditionally stable** and, if not, under what conditions is it **conditionally stable**. You may have noticed that I have written out the values of α when running each example, and that the maximum value has been $\alpha = 0.4125$ m^2/s. Let's now redo the very first example but increase $dt = 2$ such that maximum value (for copper) is now $\alpha = 0.55$ m^2/s.

```
alpha (Copper) = 0.55
alpha (Iron  ) = 0.11499999999999999
```

Out[10]:

Figure 9-4. Solutions of the 1D heat diffusion equation as in Figure 9-3, except that the value of α for the copper bar has been increased to 0.6 instead of 0.4125. Note that the copper bar simulation rapidly becomes unstable, where as the iron bar simulation remains stable ($\alpha = 0.115 m^2/s$).

So it appears that we have crossed some sort of threshold between when $\alpha = 0.4125$ m^2/s and $\alpha = 0.55$ m^2/s. To examine this, let's again look at the von Neumann stability analysis that we discussed in Module 7.

Von Neumann Stability Analysis

We want to investigate how the error difference ϵ^n between the true solution to the 1D diffusion PDE, ϕ_T and our numerical solution, ϕ, changes from time step n to $n+1$. Recall that this term is called the **amplification factor**

$$g^2 = \bar{g}g = \left| \frac{\epsilon_i^{n+1}}{\epsilon_i^n} \right|^2 \leq 1, \tag{12}$$

Again, we can assume that the error term satisfies the discretization itself the amplification factor is given by:

$$g^2 = \left| \frac{\epsilon_i^{n+1}}{\epsilon_i^n} \right|^2 = \left| \frac{\alpha \epsilon_{i+1}^n + (1-2\alpha)\,\epsilon_i^n + \alpha \epsilon_{i-1}^n}{\epsilon_i^n} \right|^2 \leq 1 \tag{14}$$

We now can assume that each term can be represented by

$$\epsilon^n = A_n e^{ikx}, \tag{15}$$

where A^n is an amplitude term at time step n such that

$$\epsilon_i^n = A_n e^{ikx}, \tag{16a}$$
$$\epsilon_{i+1}^n = A_n e^{ik(x+\Delta x)}, \tag{16b}$$
$$\epsilon_{i-1}^n = A_n e^{ik(x-\Delta x)}. \tag{16c}$$

Inserting these into equation 14 yields

$$g^2 = \left| \frac{\alpha A_n e^{ik(x+\Delta x)} + (1-2\alpha) A_n e^{ikx} + \alpha A_n e^{ik(x-\Delta x)}}{A_n e^{ikx}} \right|^2 \leq 1 \tag{17}$$

which simplifies to

$$\begin{aligned} g^2 &= \left| \alpha e^{ik\Delta x} + (1-2\alpha) + \alpha e^{-ik\Delta x} \right|^2 \leq 1 \\ &= \left| 1 + 2\alpha\,(\cos(k\Delta x) - 1) \right|^2 \leq 1 \\ &= \left| 1 - 4\alpha \sin^2\left(\frac{k\Delta x}{2} \right) \right|^2 \leq 1 \end{aligned} \tag{18}$$

where in the last step we have used the half-angle formula $2\sin^2\theta = 1 - \cos 2\theta$. Thus, the **stability criterion** for this numerical scheme is given by

$$g^2 = \left| 1 - 4\alpha \sin^2\left(\frac{k\Delta x}{2} \right) \right|^2 \leq 1. \tag{19}$$

Note that the $\sin^2$ factor ranges between 0 and 1; when it is 0 this criterion is always satisfied. Thus, we are interested in the maximal case when $\sin^2 = 1$. Accordingly, we rewrite the inequality in equation 18 as

$$g^2 = |1 - 4\alpha|^2 \leq 1, \tag{20}$$

which is satisfied in the range when $\alpha \in [0, 1/2]$. Thus, we have the following restriction on the numerical simulation:

$$\alpha = \frac{\kappa \Delta t}{\Delta x^2} < \frac{1}{2} \tag{21}$$

or for a more explicit constraint on the allowed time step Δt:

$$\Delta t < \frac{\Delta x^2}{2\kappa}, \tag{22}$$

which is consistent with the numerical simulations above. Thus, while the above FTCS approach is relatively straightforward to implement, it suffers from the severe drawback that it is only **conditionally stable**. This motivates use to look for other numerical methods that are **unconditionally stable**.

The Implicit Crank-Nicholson method

Fortunately, we do not have to look too far to find such a method. The Crank-Nicholson (CN) method represents an extension of the FTCS approach that is $\mathcal{O}\left(\Delta t^2, \Delta x^2\right)$ and **unconditionally stable**. The main idea behind this approach is that unlike the FTCS method that defines a forward-difference formula for **one unknown point** at time step $n+1$, the CN method specifies it for **three unknown points** at time step $n+1$ in the FD stencil. Figure 9-5 below illustrates the stencil associated with this numerical scheme.

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~/miniconda3/lib/python3.5/site-packages/IPython/core/formatters.py in __call__(self, obj)
    330                 pass
    331             else:
--> 332                 return printer(obj)
    333             # Finally look for special method names
    334             method = get_real_method(obj, self.print_method)

~/miniconda3/lib/python3.5/site-packages/IPython/core/pylabtools.py in <lambda>(fig)
    235
    236     if 'png' in formats:
--> 237         png_formatter.for_type(Figure, lambda fig: print_figure(fig, 'png', **kwargs))
    238     if 'retina' in formats or 'png2x' in formats:
    239         png_formatter.for_type(Figure, lambda fig: retina_figure(fig, **kwargs))

~/miniconda3/lib/python3.5/site-packages/IPython/core/pylabtools.py in print_figure(fig, fmt, bbox_inche
s, **kwargs)
    119
    120     bytes_io = BytesIO()
--> 121     fig.canvas.print_figure(bytes_io, **kw)
    122     data = bytes_io.getvalue()
    123     if fmt == 'svg':

~/miniconda3/lib/python3.5/site-packages/matplotlib/backend_bases.py in print_figure(self, filename, dpi,
facecolor, edgecolor, orientation, format, **kwargs)
   2198                     orientation=orientation,
   2199                     dryrun=True,
-> 2200                     **kwargs)
   2201                 renderer = self.figure._cachedRenderer
   2202                 bbox_inches = self.figure.get_tightbbox(renderer)

~/miniconda3/lib/python3.5/site-packages/matplotlib/backends/backend_agg.py in print_png(self, filename_o
r_obj, *args, **kwargs)
    543
    544     def print_png(self, filename_or_obj, *args, **kwargs):
--> 545         FigureCanvasAgg.draw(self)
    546         renderer = self.get_renderer()
    547         original_dpi = renderer.dpi

~/miniconda3/lib/python3.5/site-packages/matplotlib/backends/backend_agg.py in draw(self)
    462
    463         try:
--> 464             self.figure.draw(self.renderer)
    465         finally:
    466             RendererAgg.lock.release()

~/miniconda3/lib/python3.5/site-packages/matplotlib/artist.py in draw_wrapper(artist, renderer, *args, **
kwargs)
     61     def draw_wrapper(artist, renderer, *args, **kwargs):
     62         before(artist, renderer)
---> 63         draw(artist, renderer, *args, **kwargs)
     64         after(artist, renderer)
     65

~/miniconda3/lib/python3.5/site-packages/matplotlib/figure.py in draw(self, renderer)
   1149
   1150         self._cachedRenderer = renderer
-> 1151         self.canvas.draw_event(renderer)
   1152
   1153     def draw_artist(self, a):

~/miniconda3/lib/python3.5/site-packages/matplotlib/backend_bases.py in draw_event(self, renderer)
   1821         s = 'draw_event'
   1822         event = DrawEvent(s, self, renderer)
-> 1823         self.callbacks.process(s, event)
   1824
   1825     def resize_event(self):

~/miniconda3/lib/python3.5/site-packages/matplotlib/cbook.py in process(self, s, *args, **kwargs)
    552             for cid, proxy in list(six.iteritems(self.callbacks[s])):
    553                 try:
--> 554                     proxy(*args, **kwargs)
    555                 except ReferenceError:
    556                     self._remove_proxy(proxy)

~/miniconda3/lib/python3.5/site-packages/matplotlib/cbook.py in __call__(self, *args, **kwargs)
    414             mtd = self.func
    415         # invoke the callable and return the result
--> 416         return mtd(*args, **kwargs)
    417
    418     def __eq__(self, other):

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _start(self, *args)
    879
    880         # Now do any initial draw
--> 881         self._init_draw()
```

```
    882
    883         # Add our callback for stepping the animation and

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _init_draw(self)
   1538         # artists.
   1539         if self._init_func is None:
-> 1540             self._draw_frame(next(self.new_frame_seq()))
   1541
   1542         else:

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _draw_frame(self, framedata)
   1560         # Call the func with framedata and args. If blitting is desired,
   1561         # func needs to return a sequence of any artists that were modified.
-> 1562         self._drawn_artists = self._func(framedata, *self._args)
   1563         if self._blit:
   1564             if self._drawn_artists is None:

<ipython-input-4-d448627a43c3> in diffusion_1D_animate(i)
      3     k += 1
      4     ax1.clear()
----> 5     plt.plot(xx,c[:,k],'g',xx,d[:,k],'b',linewidth=3)
      6     plt.legend(['Copper','Iron'])
      7     plt.grid(True)

IndexError: index 101 is out of bounds for axis 1 with size 101
```

```
<matplotlib.figure.Figure at 0x114ab3390>
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~/miniconda3/lib/python3.5/site-packages/IPython/core/formatters.py in __call__(self, obj)
    330                 pass
    331             else:
--> 332                 return printer(obj)
    333             # Finally look for special method names
    334             method = get_real_method(obj, self.print_method)

~/miniconda3/lib/python3.5/site-packages/IPython/core/pylabtools.py in <lambda>(fig)
    235
    236     if 'png' in formats:
--> 237         png_formatter.for_type(Figure, lambda fig: print_figure(fig, 'png', **kwargs))
    238     if 'retina' in formats or 'png2x' in formats:
    239         png_formatter.for_type(Figure, lambda fig: retina_figure(fig, **kwargs))

~/miniconda3/lib/python3.5/site-packages/IPython/core/pylabtools.py in print_figure(fig, fmt, bbox_inche
s, **kwargs)
    119
    120     bytes_io = BytesIO()
--> 121     fig.canvas.print_figure(bytes_io, **kw)
    122     data = bytes_io.getvalue()
    123     if fmt == 'svg':

~/miniconda3/lib/python3.5/site-packages/matplotlib/backend_bases.py in print_figure(self, filename, dpi,
facecolor, edgecolor, orientation, format, **kwargs)
   2198                     orientation=orientation,
   2199                     dryrun=True,
-> 2200                     **kwargs)
   2201                 renderer = self.figure._cachedRenderer
   2202                 bbox_inches = self.figure.get_tightbbox(renderer)

~/miniconda3/lib/python3.5/site-packages/matplotlib/backends/backend_agg.py in print_png(self, filename_o
r_obj, *args, **kwargs)
    543
    544     def print_png(self, filename_or_obj, *args, **kwargs):
--> 545         FigureCanvasAgg.draw(self)
    546         renderer = self.get_renderer()
    547         original_dpi = renderer.dpi

~/miniconda3/lib/python3.5/site-packages/matplotlib/backends/backend_agg.py in draw(self)
    462
    463         try:
--> 464             self.figure.draw(self.renderer)
    465         finally:
    466             RendererAgg.lock.release()

~/miniconda3/lib/python3.5/site-packages/matplotlib/artist.py in draw_wrapper(artist, renderer, *args, **
kwargs)
     61     def draw_wrapper(artist, renderer, *args, **kwargs):
     62         before(artist, renderer)
---> 63         draw(artist, renderer, *args, **kwargs)
     64         after(artist, renderer)
     65

~/miniconda3/lib/python3.5/site-packages/matplotlib/figure.py in draw(self, renderer)
   1149
   1150         self._cachedRenderer = renderer
-> 1151         self.canvas.draw_event(renderer)
   1152
   1153     def draw_artist(self, a):

~/miniconda3/lib/python3.5/site-packages/matplotlib/backend_bases.py in draw_event(self, renderer)
   1821         s = 'draw_event'
   1822         event = DrawEvent(s, self, renderer)
-> 1823         self.callbacks.process(s, event)
   1824
   1825     def resize_event(self):

~/miniconda3/lib/python3.5/site-packages/matplotlib/cbook.py in process(self, s, *args, **kwargs)
    552             for cid, proxy in list(six.iteritems(self.callbacks[s])):
    553                 try:
--> 554                     proxy(*args, **kwargs)
    555                 except ReferenceError:
    556                     self._remove_proxy(proxy)

~/miniconda3/lib/python3.5/site-packages/matplotlib/cbook.py in __call__(self, *args, **kwargs)
    414             mtd = self.func
    415         # invoke the callable and return the result
--> 416         return mtd(*args, **kwargs)
    417
    418     def __eq__(self, other):

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _start(self, *args)
    879
    880         # Now do any initial draw
--> 881         self._init_draw()
```

```
    882
    883         # Add our callback for stepping the animation and

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _init_draw(self)
   1538         # artists.
   1539         if self._init_func is None:
-> 1540             self._draw_frame(next(self.new_frame_seq()))
   1541
   1542         else:

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _draw_frame(self, framedata)
   1560         # Call the func with framedata and args. If blitting is desired,
   1561         # func needs to return a sequence of any artists that were modified.
-> 1562         self._drawn_artists = self._func(framedata, *self._args)
   1563         if self._blit:
   1564             if self._drawn_artists is None:

<ipython-input-4-d448627a43c3> in diffusion_1D_animate(i)
      3     k += 1
      4     ax1.clear()
----> 5     plt.plot(xx,c[:,k],'g',xx,d[:,k],'b',linewidth=3)
      6     plt.legend(['Copper','Iron'])
      7     plt.grid(True)

IndexError: index 102 is out of bounds for axis 1 with size 101
```

```
<matplotlib.figure.Figure at 0x1149246a0>
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~/miniconda3/lib/python3.5/site-packages/IPython/core/formatters.py in __call__(self, obj)
    330                 pass
    331             else:
--> 332                 return printer(obj)
    333             # Finally look for special method names
    334             method = get_real_method(obj, self.print_method)

~/miniconda3/lib/python3.5/site-packages/IPython/core/pylabtools.py in <lambda>(fig)
    235
    236     if 'png' in formats:
--> 237         png_formatter.for_type(Figure, lambda fig: print_figure(fig, 'png', **kwargs))
    238     if 'retina' in formats or 'png2x' in formats:
    239         png_formatter.for_type(Figure, lambda fig: retina_figure(fig, **kwargs))

~/miniconda3/lib/python3.5/site-packages/IPython/core/pylabtools.py in print_figure(fig, fmt, bbox_inche
s, **kwargs)
    119
    120     bytes_io = BytesIO()
--> 121     fig.canvas.print_figure(bytes_io, **kw)
    122     data = bytes_io.getvalue()
    123     if fmt == 'svg':

~/miniconda3/lib/python3.5/site-packages/matplotlib/backend_bases.py in print_figure(self, filename, dpi,
facecolor, edgecolor, orientation, format, **kwargs)
   2198                     orientation=orientation,
   2199                     dryrun=True,
-> 2200                     **kwargs)
   2201                 renderer = self.figure._cachedRenderer
   2202                 bbox_inches = self.figure.get_tightbbox(renderer)

~/miniconda3/lib/python3.5/site-packages/matplotlib/backends/backend_agg.py in print_png(self, filename_o
r_obj, *args, **kwargs)
    543
    544     def print_png(self, filename_or_obj, *args, **kwargs):
--> 545         FigureCanvasAgg.draw(self)
    546         renderer = self.get_renderer()
    547         original_dpi = renderer.dpi

~/miniconda3/lib/python3.5/site-packages/matplotlib/backends/backend_agg.py in draw(self)
    462
    463         try:
--> 464             self.figure.draw(self.renderer)
    465         finally:
    466             RendererAgg.lock.release()

~/miniconda3/lib/python3.5/site-packages/matplotlib/artist.py in draw_wrapper(artist, renderer, *args, **
kwargs)
     61     def draw_wrapper(artist, renderer, *args, **kwargs):
     62         before(artist, renderer)
---> 63         draw(artist, renderer, *args, **kwargs)
     64         after(artist, renderer)
     65

~/miniconda3/lib/python3.5/site-packages/matplotlib/figure.py in draw(self, renderer)
   1149
   1150         self._cachedRenderer = renderer
-> 1151         self.canvas.draw_event(renderer)
   1152
   1153     def draw_artist(self, a):

~/miniconda3/lib/python3.5/site-packages/matplotlib/backend_bases.py in draw_event(self, renderer)
   1821         s = 'draw_event'
   1822         event = DrawEvent(s, self, renderer)
-> 1823         self.callbacks.process(s, event)
   1824
   1825     def resize_event(self):

~/miniconda3/lib/python3.5/site-packages/matplotlib/cbook.py in process(self, s, *args, **kwargs)
    552             for cid, proxy in list(six.iteritems(self.callbacks[s])):
    553                 try:
--> 554                     proxy(*args, **kwargs)
    555                 except ReferenceError:
    556                     self._remove_proxy(proxy)

~/miniconda3/lib/python3.5/site-packages/matplotlib/cbook.py in __call__(self, *args, **kwargs)
    414             mtd = self.func
    415         # invoke the callable and return the result
--> 416         return mtd(*args, **kwargs)
    417
    418     def __eq__(self, other):

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _start(self, *args)
    879
    880         # Now do any initial draw
--> 881         self._init_draw()
```

```
    882
    883         # Add our callback for stepping the animation and

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _init_draw(self)
   1538         # artists.
   1539         if self._init_func is None:
-> 1540             self._draw_frame(next(self.new_frame_seq()))
   1541
   1542         else:

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _draw_frame(self, framedata)
   1560         # Call the func with framedata and args. If blitting is desired,
   1561         # func needs to return a sequence of any artists that were modified.
-> 1562         self._drawn_artists = self._func(framedata, *self._args)
   1563         if self._blit:
   1564             if self._drawn_artists is None:

<ipython-input-4-d448627a43c3> in diffusion_1D_animate(i)
      3     k += 1
      4     ax1.clear()
----> 5     plt.plot(xx,c[:,k],'g',xx,d[:,k],'b',linewidth=3)
      6     plt.legend(['Copper','Iron'])
      7     plt.grid(True)

IndexError: index 103 is out of bounds for axis 1 with size 101
```

```
<matplotlib.figure.Figure at 0x114a78828>
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~/miniconda3/lib/python3.5/site-packages/IPython/core/formatters.py in __call__(self, obj)
    330                 pass
    331             else:
--> 332                 return printer(obj)
    333             # Finally look for special method names
    334             method = get_real_method(obj, self.print_method)

~/miniconda3/lib/python3.5/site-packages/IPython/core/pylabtools.py in <lambda>(fig)
    235
    236     if 'png' in formats:
--> 237         png_formatter.for_type(Figure, lambda fig: print_figure(fig, 'png', **kwargs))
    238     if 'retina' in formats or 'png2x' in formats:
    239         png_formatter.for_type(Figure, lambda fig: retina_figure(fig, **kwargs))

~/miniconda3/lib/python3.5/site-packages/IPython/core/pylabtools.py in print_figure(fig, fmt, bbox_inche
s, **kwargs)
    119
    120     bytes_io = BytesIO()
--> 121     fig.canvas.print_figure(bytes_io, **kw)
    122     data = bytes_io.getvalue()
    123     if fmt == 'svg':

~/miniconda3/lib/python3.5/site-packages/matplotlib/backend_bases.py in print_figure(self, filename, dpi,
facecolor, edgecolor, orientation, format, **kwargs)
   2198                     orientation=orientation,
   2199                     dryrun=True,
-> 2200                     **kwargs)
   2201                 renderer = self.figure._cachedRenderer
   2202                 bbox_inches = self.figure.get_tightbbox(renderer)

~/miniconda3/lib/python3.5/site-packages/matplotlib/backends/backend_agg.py in print_png(self, filename_o
r_obj, *args, **kwargs)
    543
    544     def print_png(self, filename_or_obj, *args, **kwargs):
--> 545         FigureCanvasAgg.draw(self)
    546         renderer = self.get_renderer()
    547         original_dpi = renderer.dpi

~/miniconda3/lib/python3.5/site-packages/matplotlib/backends/backend_agg.py in draw(self)
    462
    463         try:
--> 464             self.figure.draw(self.renderer)
    465         finally:
    466             RendererAgg.lock.release()

~/miniconda3/lib/python3.5/site-packages/matplotlib/artist.py in draw_wrapper(artist, renderer, *args, **
kwargs)
     61     def draw_wrapper(artist, renderer, *args, **kwargs):
     62         before(artist, renderer)
---> 63         draw(artist, renderer, *args, **kwargs)
     64         after(artist, renderer)
     65

~/miniconda3/lib/python3.5/site-packages/matplotlib/figure.py in draw(self, renderer)
   1149
   1150         self._cachedRenderer = renderer
-> 1151         self.canvas.draw_event(renderer)
   1152
   1153     def draw_artist(self, a):

~/miniconda3/lib/python3.5/site-packages/matplotlib/backend_bases.py in draw_event(self, renderer)
   1821         s = 'draw_event'
   1822         event = DrawEvent(s, self, renderer)
-> 1823         self.callbacks.process(s, event)
   1824
   1825     def resize_event(self):

~/miniconda3/lib/python3.5/site-packages/matplotlib/cbook.py in process(self, s, *args, **kwargs)
    552             for cid, proxy in list(six.iteritems(self.callbacks[s])):
    553                 try:
--> 554                     proxy(*args, **kwargs)
    555                 except ReferenceError:
    556                     self._remove_proxy(proxy)

~/miniconda3/lib/python3.5/site-packages/matplotlib/cbook.py in __call__(self, *args, **kwargs)
    414             mtd = self.func
    415         # invoke the callable and return the result
--> 416         return mtd(*args, **kwargs)
    417
    418     def __eq__(self, other):

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _start(self, *args)
    879
    880         # Now do any initial draw
--> 881         self._init_draw()
```

```
    882
    883         # Add our callback for stepping the animation and

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _init_draw(self)
   1538         # artists.
   1539         if self._init_func is None:
-> 1540             self._draw_frame(next(self.new_frame_seq()))
   1541
   1542         else:

~/miniconda3/lib/python3.5/site-packages/matplotlib/animation.py in _draw_frame(self, framedata)
   1560         # Call the func with framedata and args. If blitting is desired,
   1561         # func needs to return a sequence of any artists that were modified.
-> 1562         self._drawn_artists = self._func(framedata, *self._args)
   1563         if self._blit:
   1564             if self._drawn_artists is None:

<ipython-input-4-d448627a43c3> in diffusion_1D_animate(i)
      3     k += 1
      4     ax1.clear()
----> 5     plt.plot(xx,c[:,k],'g',xx,d[:,k],'b',linewidth=3)
      6     plt.legend(['Copper','Iron'])
      7     plt.grid(True)

IndexError: index 104 is out of bounds for axis 1 with size 101
```

```
<matplotlib.figure.Figure at 0x114ad4fd0>
```

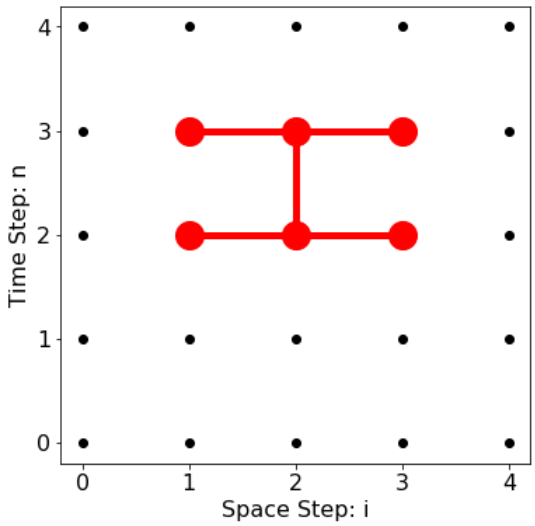


Figure 7-5. Illustration of the Crank-Nicholson stencil where known values at time step $n = 2$ are used to calculate solutions at time step $n = 3$.

As we see in the above figure, we now have a system where we have three unknown at time step $n + 1$ for each location of the stencil. However, if you shift the stencil one step to the left and right, you will also have constraint equations that can be used to help solve for the unknown forward value at $[i, n] = [2, 3]$.

CN Numerical Scheme

The CN numerical scheme is based on taking an equally weighted average of the points at time step n and $n + 1$ for approximating the second spatial derivative. We can write the CN numerical scheme in the following way:

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} = \frac{\kappa}{2}\left(\frac{\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1} + \phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n}{\Delta x^2}\right). \tag{23}$$

Again, by defining $\alpha = \frac{\kappa \Delta t}{\Delta x^2}$, we can rearrange equation 23 as

$$2\left(\phi_i^{n+1} - \phi_i^n\right) = \alpha\left(\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1} + \phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n\right), \tag{24}$$

and then move the $n + 1$ and n terms to the left- and right-hand sides, respectively:

$$-\alpha\phi_{i+1}^{n+1} + 2(1+\alpha)\phi_i^{n+1} - \alpha\phi_{i-1}^{n+1} = \alpha\phi_{i+1}^n + 2(1-\alpha)\phi_i^n + \alpha\phi_{i-1}^n. \tag{25}$$

While this seems quite a bit more complicated, we have actually a similar system in Module 8! To illustrate this, let's write this in a linear algebra framework in the standard form of $\mathbf{Ax} = \mathbf{b}$:

$$\begin{bmatrix} 2(1+\alpha) & -\alpha & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ -\alpha & 2(1+\alpha) & -\alpha & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & -\alpha & 2(1+\alpha) & -\alpha & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -\alpha & 2(1+\alpha) & -\alpha & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & -\alpha & 2(1+\alpha) & -\alpha \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & -\alpha & 2(1+\alpha) \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{I-3}^{n+1} \\ u_{I-2}^{n+1} \\ u_{I-1}^{n+1} \end{bmatrix} = \begin{bmatrix} \alpha\phi_2^n + 2(1-\alpha)\phi_1^n + \alpha\phi_0^n \\ \alpha\phi_3^n + 2(1-\alpha)\phi_2^n + \alpha\phi_1^n \\ \alpha\phi_4^n + 2(1-\alpha)\phi_3^n + \alpha\phi_2^n \\ \vdots \\ \alpha\phi_{I-2}^n + 2(1-\alpha)\phi_{I-3}^n + \alpha\phi_{I-4}^n \\ \alpha\phi_{I-1}^n + 2(1-\alpha)\phi_{I-2}^n + \alpha\phi_{I-3}^n \\ \alpha\phi_I^n + 2(1-\alpha)\phi_{I-1}^n + \alpha\phi_{I-2}^n \end{bmatrix}. \tag{26}$$

Here we see that modeling matrix $\mathbf{A}$ is a **banded matrix** on the main, super- and subdiagonal. Thus, this system of equations again represents a **triagonal system** that can be solved using standard methods of linear algebra! Also, note the first and last entries in known solution vector $\mathbf{b}$ contain the term ϕ_0^n and ϕ_I^n, respectively. These values are known and can be used to represent the **boundary conditions**.

Solution by Thomas Algorithm

Previously, we have solved systems of equations like those in equation 26 using a "brute force" method of *np.linalg.solve(Ax,bx)*, which computes the “exact” solution, $\mathbf{x}$, of the well-determined, i.e., full rank, linear matrix equation $\mathbf{Ax} = \mathbf{b}$. However, it should be pretty clear that the FD matrix $\mathbf{A}$ is actually very **sparse** and has **banded structure**. Thus, it is judicious to ask whether there are any more efficient numerical methods that can take advantage of this matrix structure.

Since you have seen the title of this subsection, you have probably surmised that the answer is yes. Let's investigate the Thomas algorithm (https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm), which can be used to solve certain tridiagonal systems (i.e., a diagonally dominant matrix (https://en.wikipedia.org/wiki/Diagonally_dominant_matrix)) defined by:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \tag{27a}$$

where $a_1 = 0$ and $c_n = 0$ and written in matrix form as

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}. \tag{28b}$$

This algorithm uses a two-step approach involving a forward and then a backward "sweep" calculation. The forward sweep consists of modifying the coefficients as follows, denoting the new coefficients with primes:

$$c'_i = \begin{cases} \dfrac{c_i}{b_i} & ; \quad i = 1 \\ \dfrac{c_i}{b_i - a_i c'_{i-1}} & ; \quad i = 2, 3, \ldots, n-1 \end{cases} \tag{29a}$$

and

$$d'_i = \begin{cases} \dfrac{d_i}{b_i} & ; \quad i = 1 \\ \dfrac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}} & ; \quad i = 2, 3, \ldots, n. \end{cases} \tag{29b}$$

The solution is then obtained by back substitution:

$$x_n = d'_n \tag{30a}$$

and

$$x_i = d'_i - c'_i x_{i+1} \qquad i = n-1, n-2, \ldots, 1. \tag{30b}$$

Let's now implement the Thomas algorithm using two different functions. The first function *TDMAsolver(a,b,c,d)* implements the Thomas Algorithm solution, while the second function *Solve_Tridiagonal_homogeneous_Thomas(a,b,c,d)* sets up the coefficients to be passed to the *TDMAsolver*.

Let's now redo the scenario from above that proved to be unstable with $dt = 2$ and $\alpha = 0.55\text{m}^2/\text{s}$, but go back to simple Dirchelet boundary conditions where $\phi(x = 0, t) = \phi(x = 1, t) = 0$.

```
alpha (Copper) = 0.55
alpha (Iron  ) = 0.11499999999999999
```

Out[17]:

0:00 / 0:03

Figure 9-6. Movie showing the same example as in Figure 9-4, but using a Crank-Nicholson solver with dt=2.0s. We see that the CN approach does not suffer from any (apparent) numerical instabilities.

Let's now rerun the solution above, but increase the time step to, say, $\Delta t = 20$s:

```
alpha (Copper) = 5.500000000000001
alpha (Iron  ) = 1.15
```

Out[18]:

0:00 / 0:03

Figure 9-7. Movie showing the same example as in Figure 9-4, but using a Crank-Nicholson solver with Δt=20.0s. We see that the CN approach does not suffer from any (apparent) numerical instabilities.

Importance of CN method

There are a number of different factors that have come to light in this series of investigations

- **Unconditional Stability** - The CN method allows you to run simulations that are guaranteed to be stable, which can be shown to be the following through a von Neumann staility analysis for all values of α:

$$g^2 = \left| \frac{1 - 2\alpha \sin^2(\gamma/2)}{1 + 2\alpha \sin^2(\gamma/2)} \right|^2 \leq 1 \tag{31}$$

- **Large time-stepping** - We have seen that we can use large Δt time steps, which means that it is much quicker to forward simulate the numerical solution to the steady state solution.
- **Reduced computational complexity** - The computational complexity of the Thomas Algorithm is $\mathcal{O}(nx)$ compared to $O(nx^3)$ for Gaussian elimination, which means that this approach scales much better as nx gets very large.

Solving the 1D Convection-Diffusion equation

The 1D convection-diffusion equation for a convective velocity v_x and diffusivity κ is given by:

$$\frac{\partial \phi}{\partial t} + v_x \frac{\partial \phi}{\partial x} = \kappa \frac{\partial^2 \phi}{\partial x^2} \tag{32}$$

which is subject to some initial condition

$$\phi(x, t = 0) = \phi_0(x). \tag{33}$$

Here, both the convection velocity v_x and diffusivity κ are assumed to be constant.

Crank-Nicholson Numerical Approach

Let's take the Crank-Nicholson approach that we learned in the section above and apply it to the the 1D convection-diffusion equation. This leads to the following discretized equation

$$\frac{\phi_i^{n+1} - \phi_i^n}{\Delta t} + \frac{v_x}{2}\left(\frac{\phi_{i+1}^{n+1} - \phi_{i-1}^{n+1}}{2\Delta x} + \frac{\phi_{i+1}^{n} - \phi_{i-1}^{n}}{2\Delta x} \right) = \frac{\kappa}{2}\left(\frac{\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1} + \phi_{i+1}^{n} - 2\phi_i^{n} + \phi_{i-1}^{n}}{\Delta x^2} \right) \tag{34}$$

Let's now multiply through by Δt

$$\phi_i^{n+1} - \phi_i^n + \frac{v_x \Delta t}{4\Delta x}\left(\phi_{i+1}^{n+1} - \phi_{i-1}^{n+1} + \phi_{i+1}^n - \phi_{i-1}^n\right) = \frac{\kappa \Delta t}{2\Delta x^2}\left(\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1} + \phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n\right), \tag{35a}$$

and use the following subsitutions of the Courant number $C = \frac{v_x \Delta t}{\Delta x}$ and the diffusion number $\alpha = \frac{\kappa \Delta t}{\Delta x^2}$ and multiply through by 4 such that

$$4\phi_i^{n+1} - 4\phi_i^n + C\left(\phi_{i+1}^{n+1} - \phi_{i-1}^{n+1} + \phi_{i+1}^n - \phi_{i-1}^n\right) = 2\alpha\left(\phi_{i+1}^{n+1} - 2\phi_i^{n+1} + \phi_{i-1}^{n+1} + \phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n\right). \tag{35b}$$

Let's now bring all of the $n+1$ terms to the left and all of the n terms to the right:

$$4\phi_i^{n+1} + C\phi_{i+1}^{n+1} - C\phi_{i-1}^{n+1} - 2\alpha\phi_{i+1}^{n+1} + 4\alpha\phi_i^{n+1} - 2\alpha\phi_{i-1}^{n+1} = 4\phi_i^n - C\phi_{i+1}^n + C\phi_{i-1}^n + 2\alpha\phi_{i+1}^n - 4\alpha\phi_i^n + 2\alpha\phi_{i-1}^n. \tag{35c}$$

Now let's collect like terms to yield

$$(C - 2\alpha)\,\phi_{i+1}^{n+1} + 4(1+\alpha)\,\phi_i^{n+1} - (C + 2\alpha)\,\phi_{i-1}^{n+1} = -(C - 2\alpha)\,\phi_{i+1}^n + 4(1-\alpha)\,\phi_i^n + (C + 2\alpha)\,\phi_{i-1}^n. \tag{35d}$$

Again, we see that even though the resulting discretized system is more complex, it is still represented by a tridiagonal system!

Let's look at the resulting system in a linear algebra framework

$$\begin{bmatrix} 4(1+\alpha) & (C-2\alpha) & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ -(C+2\alpha) & 4(1+\alpha) & (C-2\alpha) & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & -(C+2\alpha) & 4(1+\alpha) & (C-2\alpha) & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -(C+2\alpha) & 4(1+\alpha) & (C-2\alpha) & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & -(C+2\alpha) & 4(1+\alpha) & (C-2\alpha) \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & -(C+2\alpha) & 4(1+\alpha) \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{I-3}^{n+1} \\ u_{I-2}^{n+1} \\ u_{I-1}^{n+1} \end{bmatrix} \tag{36}$$

$$= \begin{bmatrix} -(C-2\alpha)\,\phi_2^n + 4(1-\alpha)\,\phi_1^n + (C+2\alpha)\,\phi_0^n \\ -(C-2\alpha)\,\phi_3^n + 4(1-\alpha)\,\phi_2^n + (C+2\alpha)\,\phi_1^n \\ -(C-2\alpha)\,\phi_4^n + 4(1-\alpha)\,\phi_3^n + (C+2\alpha)\,\phi_2^n \\ \vdots \\ -(C-2\alpha)\,\phi_{I-2}^n + 4(1-\alpha)\,\phi_{I-3}^n + (C+2\alpha)\,\phi_{I-4}^n \\ -(C-2\alpha)\,\phi_{I-1}^n + 4(1-\alpha)\,\phi_{I-2}^n + (C+2\alpha)\,\phi_{I-3}^n \\ -(C-2\alpha)\,\phi_I^n + 4(1-\alpha)\,\phi_{I-1}^n + (C+2\alpha)\,\phi_{I-2}^n \end{bmatrix}.$$

```
diffusion #: alpha (fast) = 100.0
diffusion #: alpha (slow) = 1.0
Courant   #: C = 0.2
```

Out[21]:

Solving the 2D Convection-Diffusion Equation

Let's now look at expanding our dimensionality to cover the 2D convection-diffusion equation. This equation is given in general by the following:

$$\frac{\partial \phi}{\partial t} + \mathbf{v} \cdot \nabla \phi = \kappa \nabla^2 \phi, \tag{37a}$$

and with specific Cartesian partial derivatives

$$\frac{\partial \phi}{\partial t} + v_x \frac{\partial \phi}{\partial x} + v_y \frac{\partial \phi}{\partial y} = \kappa \left(\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \right). \tag{37b}$$

Given that we (1) know how to solve the 1D convection-diffusion equation from above and (2) used an alternating direction implict (ADI) solver in Module 8, we might expect that their combination presents us with a two-step procedure to solve the 2D convection-diffusion equation

Step 1 - First half-step in x direction

The first half step provides the following update in the x direction:

$$\begin{aligned} &\frac{\phi_{i,j}^{n+1/2} - \phi_{i,j}^{n}}{\Delta t/2} + v_x \left(\frac{\phi_{i+1,j}^{n+1/2} - \phi_{i-1,j}^{n+1/2}}{2\Delta x} \right) + v_y \left(\frac{\phi_{i,j+1}^{n} - \phi_{i,j-1}^{n}}{2\Delta y} \right) \\ &= \kappa \left(\frac{\phi_{i+1,j}^{n+1/2} - 2\phi_{i,j}^{n+1/2} + \phi_{i-1,j}^{n+1/2}}{\Delta x^2} + \frac{\phi_{i,j+1}^{n} - 2\phi_{i,j}^{n} + \phi_{i,j-1}^{n}}{\Delta y^2} \right) \end{aligned} \tag{38}$$

Multiplying through by $2\Delta t$ and then using $C_x = \frac{v_x \Delta t}{\Delta x}$, $C_y = \frac{v_y \Delta t}{\Delta y}$, $\alpha_x = \frac{\kappa \Delta t}{\Delta x^2}$, and $\alpha_y = \frac{\kappa \Delta t}{\Delta y^2}$ leads to

$$\begin{aligned} 4\phi_{i,j}^{n+1/2} - 4\phi_{i,j}^{n} + C_x \left(\phi_{i+1,j}^{n+1/2} - \phi_{i-1,j}^{n+1/2} \right) + C_y \left(\phi_{i,j+1}^{n} - \phi_{i,j-1}^{n} \right) = 2\alpha_x \left(\phi_{i+1,j}^{n+1/2} - 2\phi_{i,j}^{n+1/2} + \phi_{i-1,j}^{n+1/2} \right) \\ + 2\alpha_y \left(\phi_{i,j+1}^{n} - 2\phi_{i,j}^{n} + \phi_{i,j-1}^{n} \right) \end{aligned} \tag{39}$$

Collecting like terms and moving those dependent on time step $n + 1/2$ to the left-hand side and n to the right-hand side results in

$$(C_x - 2\alpha_x)\phi_{i+1,j}^{n+1/2} + 4(1 + \alpha_x)\phi_{i,j}^{n+1/2} - (C_x + 2\alpha_x)\phi_{i-1,j}^{n+1/2} = -(C_y - 2\alpha_y)\phi_{i,j+1}^{n} + 4(1 - \alpha_y)\phi_{i,j}^{n} + (C_y + 2\alpha_y)\phi_{i,j-1}^{n} \tag{40}$$

which is a tridiagonal matrix as is expected.

Step 2 - Second half-step in y direction

The second half step provides the following update in the y direction:

$$\begin{aligned} &\frac{\phi_{i,j}^{n+1} - \phi_{i,j}^{n+1/2}}{\Delta t/2} + v_x \left(\frac{\phi_{i+1,j}^{n+1/2} - \phi_{i-1,j}^{n+1/2}}{2\Delta x} \right) + v_y \left(\frac{\phi_{i,j+1}^{n+1} - \phi_{i,j-1}^{n+1}}{2\Delta y} \right) \\ &= \kappa \left(\frac{\phi_{i+1,j}^{n+1/2} - 2\phi_{i,j}^{n+1/2} + \phi_{i-1,j}^{n+1/2}}{\Delta x^2} + \frac{\phi_{i,j+1}^{n+1} - 2\phi_{i,j}^{n+1} + \phi_{i,j-1}^{n+1}}{\Delta y^2} \right) \end{aligned} \tag{41}$$

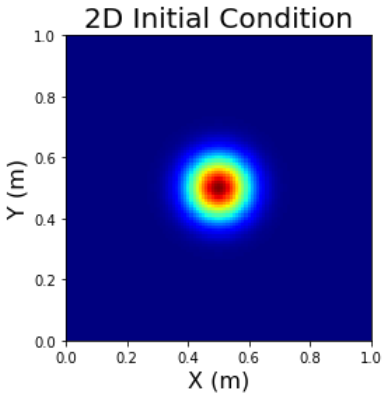
Multiplying through by $2\Delta t$ and again using $C_x = \frac{v_x \Delta t}{\Delta x}$, $C_y = \frac{v_y \Delta t}{\Delta y}$, $\alpha_x = \frac{\kappa \Delta t}{\Delta x^2}$, and $\alpha_y = \frac{\kappa \Delta t}{\Delta y^2}$ leads to

$$\begin{aligned} 4\phi_{i,j}^{n+1} - 4\phi_{i,j}^{n+1/2} + C_x \left(\phi_{i+1,j}^{n+1/2} - \phi_{i-1,j}^{n+1/2} \right) + C_y \left(\phi_{i,j+1}^{n+1} - \phi_{i,j-1}^{n+1} \right) = 2\alpha_x \left(\phi_{i+1,j}^{n+1/2} - 2\phi_{i,j}^{n+1/2} + \phi_{i-1,j}^{n+1/2} \right) \\ + 2\alpha_y \left(\phi_{i,j+1}^{n+1} - 2\phi_{i,j}^{n+1} + \phi_{i,j-1}^{n+1} \right) \end{aligned} \tag{42}$$

Collecting like terms and moving those dependent on time step n to the left-hand side and $n + 1/2$ to the right-hand side results in

$$(C_y - 2\alpha_y)\phi_{i,j+1}^{n+1/2} + 4(1 + \alpha_y)\phi_{i,j}^{n+1/2} - (C_y + 2\alpha_y)\phi_{i,j-1}^{n+1/2} = -(C_x - 2\alpha_x)\phi_{i+1,j}^{n} + 4(1 - \alpha_x)\phi_{i,j}^{n} + (C_x + 2\alpha_x)\phi_{i-1,j}^{n} \tag{43}$$

which, again, is a tridiagonal matrix as is expected.



Out[25]:

0:00 / 0:05