Workshop: Creating faster Matlab programs for simple differential equations

Jared Barber

February 19, 2011
Background, Classes: Double

Four classes for variables

• Class *Double*-Contain numbers
  – Vectors
    • Row vector >> \( u = [0, 1, 2.5] \)
    • Column vector >> \( u = [0;1;2.5] \)
  – Matrices >> \( M = [1,2;3,4]; \)
  – N-dimensional arrays (e.g. \( x,y,z,t \))
    • >> \( N = \text{ones}(3,3,3,4); \)
    • >> \( N = \text{ndgrid}(1:3,1:3,1:3,1:4); \)
Classes: String

Four classes for variables

• Class *String*-List of characters (no numerical value associated with them)
  – Initializing
    • >> str1 = ‘Hello’;
    • >> str2 = ‘my name is’;
  – Concatenating
    • >> str3 = [str1,’,’str2];
  – Selecting only a few
    • >> str4 = str2(1:4);
Classes: Cells

Four classes for variables

• Class *Cell*-Array or list where each entry can be pretty much anything
  
  – >> cell1 = cell(1,4);
  – >> cell1{1} = 1000;
  – >> cell1{2} = ‘Hello’;
  – >> cell1{3} = [1,2;3,4];
  – >> cell1{4} = {1000,’Hello’,[1,2;3,4]};
Classes: Structures

Four classes for variables
• Class *Structure*- Stores multiple variables, and associated names, at once
  – Initializing-dynamic memory allocation
    • >> struc1.parameter1 = 1
    • >> struc1.parameter2 = 3.14
    • >> struc1.parameter3 = ‘off’
  – Parameter1, parameter2, parameter3 are “fields” that store anything you want
  – Initializing-static memory allocation
    • >> struc2 =
    struct('parameter1',1,'parameter2',3.14,'parameter3','off')
Classes: Summary

When are the four types useful

• Double-used for computations and main class used

• Strings-can use for output (e.g. plots) and for dynamic structure allocation/storage
  – Struc1.(‘parameter1’) = 1 does exact same thing as Struc1.parameter1 = 1

• Cells-can use for storage, especially when collecting solutions with the same name

• Structure-used for storage and passing a lot of parameters from one function to another
Background, solving an ode: The ode

- Consider the following initial value problem:
  \[ y' + y = e^{-t^2} \sin(t); \ y(0) = 1. \]
- To solve this equation using matlab, we first rewrite the diff eq in normal form:
  \[ y' = -y + e^{-t^2} \sin(t) \]
- We then write an m-file that, when given t and y, returns the value of the rhs of this equation.
Solving an ode: m-file for rhs

At command prompt type:
>> edit my_rhs.m
This should open the editor. Then write the function inside the editor:
• function rhs = my_rhs(t,y)
  • rhs = -y + exp(-t.^2).*sin(t);
• end
(Note: the end is optional.)
After you’ve typed this in, save the program as my_rhs.m.
Solving an ODE: Using ode45

• As a reminder we are solving:
  \[ y' + y = e^{-t^2} \sin(t); \quad y(0) = 1. \]

• Now that we saved the file we type
  >> [ts,ys] = ode45(@my_rhs,[0,10],1);

• Where we’ve written [ts,ys] =
  ode45(function_handle, [t_initial,t_final], y(0 ));

• To see the solution we plot
  – >> plot(ts,ys);
Solving and ODE: Looping through ICs

• Consider multiple initial conditions
• >> edit ode_mult_ics.m
• Type in the program
• function ode_mult_ics
  
  ics = linspace(0.1,2,10);
  
  for icc = 1:length(ics)
    [ts,ys] = ode45(@my_rhs,[0,10],ics(icc));
    plot(ts,ys);
    hold on;
  end
  
• end
• Save and run
Using different classes to store solutions: Dynamic vs Static memory allocation with matrices

• Matlab allows for both static and dynamic memory allocation of matrices
  – Static allocation (generally faster)
    • Initialize: >> M = ones(2,2);
    • Adjust each entry (without changing the size of the matrix): >> M(1,1) = 1, M(1,2) = 4, M(2,1) = 3, M(2,2) = -1;
  – Dynamic allocation (generally slower)
    • No initialization
    • M(1,1) = 1, M(1,2) = 4, M(2,1) = 3, M(2,2) = -1
Storing solutions: Dynamic matrix allocation (the wrong way)

Resave prog as ode_mult_ics2, then type

- function ode_mult_ics2
- 
- ics = linspace(0.1,2,10);
- for icc = 1:length(ics)
-     [ts_temp,ys_temp] = ode45(@my_rhs,[0,10],ics(icc));
-     ts(:,icc) = ts_temp;
-     ys(:,icc) = ys_temp;
-     plot(ts(:,icc),ys(:,icc));
-     hold on;
- end
- 
- end

Save and run
Storing Solutions: Errors and debugging aside

- Can turn on debugging in multiple ways:
  - Debug->Stop if Errors/Warnings->Always stop if error (can also stop if warning...e.g. warnings division by zero)
  - Click on left hand side of program-F5 to run-F10 to “step through” program—includes conditional breaks
  - Actually enter “keyboard” into the program

- To exit a program when in debug mode
  - Shift-F5 when in editor window
  - >> dbquit when at command prompt
Storing Solutions: Errors and debugging aside

• Can also look at M-lint
  – Right hand side of editor
  – Scroll mouse over it to see what errors matlab thinks you have (red color)
  – Scroll mouse over it to see what suggestions matlab has for you (e.g. here dynamic vs static allocation)
Storing solutions: Static matrix allocation

Resave prog as ode_mult_ics3, then type

- \texttt{function \ ode\_mult\_ics3}

- \texttt{ics = linspace(0.1,2,10);}
- \texttt{sol\_ts = linspace(0,10,100);}
- \texttt{for icc = 1:length(ics)}
- \quad \texttt{[ts\_temp,ys\_temp] = ode45(@my\_rhs,sol\_ts,ics(icc));}
- \quad \texttt{ts(:,icc) = ts\_temp;}
- \quad \texttt{ys(:,icc) = ys\_temp;}
- \quad \texttt{end}

- \texttt{plot(ts,ys);}

- \texttt{end}

Save and run
Storing solutions: Static cell allocation

Don’t need to save all 100 points, just save the ones ode45 thinks are necessary

• function ode_mult_ics4
  • ics = linspace(0.1,2,10);
  • ts = cell(1,10); ys = cell(1,10);
  • for icc = 1:length(ics)
    •   [ts_temp,ys_temp] = ode45(@my_rhs,[0,10],ics(icc));
    •   ts{icc} = ts_temp;
    •   ys{icc} = ys_temp;
    • end
  •
• for icc = 1:length(ics)
  •   plot(ts{icc},ys{icc});
  •   hold on
  • end
• end

Save as ode_mult_ics4 and run.
Storing solutions: Dynamic structure allocation

Don’t need to save all 100 points, just save the ones ode45 thinks are necessary

```
function ode_mult_ics5

ics = linspace(0.1,2,10);

for icc = 1:length(ics)
    [ts_temp,ys_temp] = ode45(@my_rhs,[0,10],ics(icc));
    sol(icc).ts = ts_temp;
    sol(icc).ys = ys_temp;
end

for icc = 1:length(ics)
    plot(sol(icc).ts,sol(icc).ys);
    hold on
end
end
```

Save as ode_mult_ics5 and run. Note: EVERYTHING is stored inside the single variable sol (a structure).
Storing solutions: Summary

• Can store inside
  – Matrix-care must be taken as all rows and columns must be of same size
  – Cell-Easier when receiving data of different lengths
  – Structure-Takes larges amount of data and stores them in one place
Store Solutions: Timing your program/computer

- How fast is your computer relative to other ones?
  - bench

- How long is your program taking?
  - tic; ode_mult_ics4; toc

- What’s taking the most time in this program?
  - Desktop -> Profiler

- Important note: To get accurate timing...execute the given command (bench, tic/toc, profiler) multiple times, not just once...first time always is slow
Store Solutions: Timing your program/computer

• Desktop -> Profiler
  – Type ode_mult_ics4 into “Run this code” box
  – See general list of all progs called at any time
  – Click on ode_mult_ics4

• What’s taking the most time in this program?
  – Actual integration
  – Plotting
  – Allocation not that much time (this is because our variables are relatively small and easier to store).
Store Solutions: Timing your program/computer

• Can use profiler and tic/toc to decide between two different ways of doing things (e.g. ode_mult_ics3 with matrix storage vs ode_mult_ics4 with cell storage)

• Results are platform dependent
Solving an ode with parameters: Adjusted ODE

• Consider instead

\[ y' + ay = be^{-ct^2} \sin(\omega t) \]

• That is, we have many parameters we wish to feed in/adjust as we see fit

• Our equation in normal form becomes:

\[ y' = -ay + be^{-ct^2} \sin(\omega t) \]
Solving an ode with parameters: m-file for rhs

At command prompt type:

```matlab
>> edit my_rhs2.m
```

Then write the function inside the editor:

- `function rhs = my_rhs2(t,y,a,b,c,omega)
- ```matlab
  rhs = -a.*y+b.*exp(-c.*t.^2).*sin(omega.*t);
- end
  ```

After you’ve typed this in, save the program as my_rhs2.m.
Solving an ODE with parameters: New looping function

Then we save ode_mult_ics5.m as ode2_mult_ics.m

```
function ode2_mult_ics

ics = linspace(0.1,2,10);
a = 1; b = 1; c = 1; omega = 1;
for icc = 1:length(ics)
    [ts_temp,ys_temp] = ...
        ode45(@(t,y) my_rhs2(t,y,a,b,c,omega),[0,10],ics(icc));
    sol(icc).ts = ts_temp;
    sol(icc).ys = ys_temp;
end

for icc = 1:length(ics)
    plot(sol(icc).ts,sol(icc).ys);
    hold on
end
end
```

Save as ode2_mult_ics and run.
Solving an ode with parameters: m-file for rhs-with a structure

At command prompt type:
>> edit my_rhs3.m

Then write the function inside the editor:
•  function  rhs = my_rhs3(t,y,paras)
•       rhs = -paras.a.*y+paras.b.*...
•       exp(-paras.c.*t.^2).*sin(paras.omega.*t);
•    end

After you’ve typed this in, save the program as my_rhs3.m.
Solving an ODE with parameters: New looping function-with a structure

Then we save ode2_mult_ics.m as ode2_mult_ics2.m

```matlab
function ode2_mult_ics2

ics = linspace(0.1,2,10);
paras.a = 1; paras.b = 1; paras.c = 1; paras.omega = 1;
for icc = 1:length(ics)
    [ts_temp,ys_temp] = ... 
    ode45(@(t,y) my_rhs3(t,y,paras),[0,10],ics(icc));
    sol(icc).ts = ts_temp;
    sol(icc).ys = ys_temp;
end

for icc = 1:length(ics)
    plot(sol(icc).ts,sol(icc).ys);
    hold on
end
end
```

Save as ode2_mult_ics2 and run.
Solving an ODE with parameters: Feeding in your own parameters, varargin

- Consider the program:
- \textbf{function} \texttt{varargin\_ex(varargin)}
- \hspace{1em} \texttt{disp(varargin)};
- \texttt{end}

- Matlab takes whatever inputs you feed in, stores them in the cell \texttt{varargin}, then prints out the contents of that cell. Try:

\begin{verbatim}
>> varargin\_ex(1)
>> varargin\_ex(‘hello’)
>> varargin\_ex(‘hello’,2)
\end{verbatim}
Solving an ODE with parameters : Feeding in your own parameters-varargin

function sol = ode2_mult_ics3(varargin)
•
• ics = linspace(0.1,2,10);
• paras.a = 1; paras.b = 1; paras.c = 1; paras.omega = 1;
• for vac = 1:2:length(varargin)
•   paras.(varargin{vac}) = varargin{vac+1};
• end
• for icc = 1:length(ics)
•   [ts_temp,ys_temp] = ...
•   ode45(@(t,y) my_rhs3(t,y,paras),[0,10],ics(icc));
•   sol(icc).ts = ts_temp;
•   sol(icc).ys = ys_temp;
• end

• for icc = 1:length(ics)
•   plot(sol(icc).ts,sol(icc).ys);
•   hold on
• end

• end

The program sets default values for our parameters and rewrites them if the user feeds in the right data.
Solving an ODE with parameters: New looping function-with a structure

• To see it work, try:

```>> close all; figure(1); sol = ode2_mult_ics3('a',1);
This should give you your first result;

>> figure(2); sol = ode2_mult_ics3('a',2);
More decay.

>> figure(3); sol = ode2_mult_ics3('c',0);
No exponential decay of source term.
```
Solving an ODE with parameters: Varargin Summary

• Varargin with default values can be combined with loops outside of the called function to quickly obtain multiple results for “what if” scenarios...particularly what happens if you change one (or multiple) of your parameter values by a certain amount.

• It can also be used to turn on and off certain features of the program. E.g. you may want ode2_mult_ics3 to produce plots in some situation, but not others.

• I have found that this sort of adjusting of parameters from the command line is most easily accomplished if your parameters are stored in a structure but can still be done if parameters are stored in something else, e.g. a cell.
Parfor loop: Turn off plotting in most recent function

• function sol = ode2_mult_ics4(varargin)
  
  • ics = linspace(0.1,2,10);
  paras.a = 1; paras.b = 1; paras.c = 1; paras.omega = 1;
  for vac = 1:2:length(varargin)
    paras.(varargin{vac}) = varargin{vac+1};
  end
  for icc = 1:length(ics)
    [ts_temp,ys_temp] = ...
    ode45(@(t,y) my_rhs3(t,y,paras),[0,10],ics(icc));
    sol(icc).ts = ts_temp;
    sol(icc).ys = ys_temp;
  end
  % No plotting
  end
Parfor loop: Comparison between normal for loop and parfor loop

• First, try without parfor:
  • as = linspace(0,2,32);
  • tic;
  • for ac = 1:length(as)
    • sols(ac).sol = ode2_mult_ics4('a',as(ac));
  • end
  • cput1 = toc

• Next try with parfor: Open up 3 “matlab workers”
  >> matlabpool 3
  • Then use parfor loop...just replace for with parfor: (You may have to do this twice)
  • tic;
  • parfor ac = 1:length(as)
    • sols(ac).sol = ode2_mult_ics4('a',as(ac));
  • end
  • cput2 = toc
  • relative_rates = cput1/cput2
Parfor loop: Summary 1

• You can use any number of cores...up to the maximum available on your machine: maxNumCompThreads

• Suggestion: Keep at least 1 core open for normal computation (e.g. using internet explorer) unless you plan on running it overnight

• There seems to be a memory limit per core and if exceeded matlab can freeze (solution, don’t use so many points in your grid, or don’t use parfor)
Parfor loops: Summary 2

• They do not guarantee a speed up of nx where n is the number of cores (e.g., see above)

• How successful they are depends on
  – the application...in some applications they will only slow down your code
  – Your machine architecture
  – Memory demand/processing demand—lower memory and higher processing demands give better parfor loops
Parfor loops: Summary 3

- Parfor loops cannot always replace for loops
- Parfor loops do not always give the same results as for loops
- Guidelines:
  - Try not to have one loop iteration depend on any other (though MATLAB can sometimes handle this situation with a parfor loop)
  - Do not nest parfor loops...try to keep them as far outside as possible (parfor, for, for, stuff, end, end, end, end not for, for, parfor, stuff, end, end)
  - Avoid plotting...the plots will not show up inside of a parfor loop but will slow down computation
- Final note: I do get around 5-6x speedup for my application on an effectively 8-core machine
PDE: The advection equation

- Consider the advection equation in two-dimensions:
  \[ u_t + au_x = 0; u(x,0) = u_0(x) \]
- Where \( a \) is a constant. Assuming infinite boundaries the solution is given by:
  \[ u(x,t) = u_0(x - at) \]
- The pde describes shapes/functions/waves that move to the right at constant velocity \( a \).
PDE: Burger’s equation

• Consider Burger’s equation in two-dimensions:

\[ u_t + uu_x = u_t + (u^2 / 2)_x = 0; \]

\[ u(x,0) = -(1 - x^2)xe^{-x^2} \]
\[ u(1,t) = u(-1,t) = 0 \]

• There are many different ways to discretize and solve this pde. We focus in on two.
PDE: Burger’s equation—Lax-Friedrichs’ Method

- Consider Burger’s equation in two-dimensions:

\[
0 = u_t + \left( \frac{u^2}{2} \right)_x \approx \\
\frac{u_{i}^{n+1} - \left( u_{i-1}^{n} + u_{i+1}^{n} \right) / 2}{dt} + \frac{(u_{i+1}^{n} / 2)^2 - (u_{i-1}^{n} / 2)^2}{2dx} \\
\frac{u_{i}^{n+1} = \left( u_{i-1}^{n} + u_{i+1}^{n} \right) / 2}{2} - \frac{dt \left( u_{i+1}^{n} \right)^2 - \left( u_{i-1}^{n} \right)^2}{dx} \frac{4}{4}
\]
PDE: Burger’s equation—Lax-Friedrichs’ Method

- function [xsM,tsM,us,cput] = integrate_wave_LF(varargin)
- paras.nx = 101; paras.nt = 101; paras.tf = 0.5;
- for vac = 1:2:length(varargin)
  - paras.(varargin{vac}) = varargin{vac+1};
- end
- paras.dx = 2/(paras.nx-1); paras.dt = paras.tf/(paras.nt-1);
- xs = linspace(-1,1,paras.nx);
- u0 = -(1-xs.^2).*xs.*exp(-xs.^2);
- ts = linspace(0,paras.tf,paras.nt);
- us = zeros(paras.nt,paras.nx);
- us(1,:) = u0;
- xl = length(xs);
- tl = length(ts);
- tic;
- for tc = 2:tl
  - for uc = 2:xl-1
    - us(tc,uc) = (us(tc-1,uc+1)+us(tc-1,uc-1))./2-...
    - (paras.dt./paras.dx./4).*...
    - (us(tc-1,uc+1).^2-us(tc-1,uc-1).^2);
  - end
- end
- cput = toc;
- [xsM,tsM] = meshgrid(xs,ts);
- end
PDE: Function for plotting solution

The following function allows us to take the solution from the previous program and plot it.

```matlab
function plot_pde(xsM,tsM,us)
    close(figure(2)); figure(2);
    paras.dt = diff(tsM(1:2,1));
    for tc = round(linspace(1,size(tsM,1),100))
        plot(xsM(tc,:),us(tc,:));
        ylim([min(us(:)),max(us(:))]);
        pause(paras.dt)
    end
end
```

Hence, to plot the solution, first get the solution by running:

```matlab
>>[xsM,tsM,us] = integrate_wave_LF;
>>plot_pde(xsM,tsM,us)
```

For more refinement try:

```matlab
>>[xsM,tsM,us] = integrate_wave_LF('nx',2001,'nt',2001); plot_pde(xsM,tsM,us)
```

To run longer try:

```matlab
>>[xsM,tsM,us] = integrate_wave_LF('tf',1,'nx',2001,'nt',2001); plot_pde(xsM,tsM,us)
```
PDE: Burger’s equation: LF Method, Vectorizing Code, Vectorizing Code Usually makes code run faster

```matlab
function [xsM, tsM, us, cput] = integrate_wave_LF_vectorized(varargin)

paras.nx = 101; paras.nt = 101; paras.tf = 0.5;
for vac = 1:2:length(varargin)
    paras.(varargin{vac}) = varargin{vac+1};
end
paras.dx = 2/(paras.nx-1); paras.dt = paras.tf/(paras.nt-1);
xs = linspace(-1,1,paras.nx);
u0 = -(1-xs.^2).*xs.*exp(-xs.^2);
ts = linspace(0,paras.tf,paras.nt);
us = zeros(paras.nt,paras.nx);
us(1,:) = u0;
tl = length(ts);
tic;
for tc = 2:tl
    us(tc,2:end-1) = (us(tc-1,3:end)+us(tc-1,1:end-2))./2-
    (paras.dt./paras.dx./4).*
    (us(tc-1,3:end).^2-us(tc-1,1:end-2).^2);
end
cput = toc;
[xsM,tsM] = meshgrid(xs,ts);
end
```
PDE: Checking and comparing vectorized result

• First, make sure code produces same results by trying:
  
  ```
  >> [xsM,tsM,us] = integrate_wave_LF_vectorized;
  plot_pde(xsM,tsM,us)
  >> [xsM,tsM,us] =
      integrate_wave_LF_vectorized('tf',1,'nx',2001,'nt',2001);
  plot_pde(xsM,tsM,us)
  ```

• Second, compare the two codes and see which one takes longer:

  ```
  >> [xsM,ysM,us,cpu_t] = integrate_wave_LF; cpu_t
  >> [xdM,ysM,us,cpu_t] = integrate_wave_LF_vectorized; cpu_t
  ```
PDE: Vectorizing Code in General

• Everyone suggests you vectorize code as it *usually* leads to more efficient code

• This is a rare case where direct vectorization doesn’t help out. Why?

• Accessing via indices can cost time: e.g. `us(4,2:end-1)`
  – This cost effectively decreases (relative to the for-loop approach) as the size of the matrices increases
  – The cost depends on the way you index
PDE: Comparing vectorized code for larger matrices

• Compare the two codes for a larger number of values:

>>> [xsM,ysM,us,cpu_t] = integrate_wave_LF('nx',2001,'nt',2001); cpu_t

>>> [xsM,ysM,us,cpu_t] = integrate_wave_LF_vectorized('nx',2001,'nt',2001); cpu_t

• Their times are getting closer
PDE: Burger’s equation: LF Method, Vectorizing Code, 
Try using Columns instead

function [xsM,tsM,us,cput] = integrate_wave_LF_vectorized_cols(varargin)
paras.nx = 101; paras.nt = 101; paras.tf = 0.5;
for vac = 1:2:length(varargin)
    paras.(varargin{vac}) = varargin{vac+1};
end
paras.dx = 2/(paras.nx-1); paras.dt = paras.tf/(paras.nt-1);
xs = linspace(-1,1,paras.nx);
u0 = -(1-xs.^2).*xs.*exp(-xs.^2);
us = zeros(paras.nt,paras.nx);
u0(1,:) = u0;
tl = length(ts);
tic;
us = us';
for tc = 2:tl
    us(2:end-1,tc) = (us(3:end,tc-1)+us(1:end-2,tc-1))./2-...
        (paras.dt./paras.dx./4).*... 
        (us(3:end,tc-1).^2-us(1:end-2,tc-1).^2);
end
us = us';
cput = toc;
[xsM,tsM] = meshgrid(xs,ts);
end
PDE: Comparing using columns rather than rows

• Compare the three codes for a larger number of values:

```matlab
>> [xsM,ysM,us,cpu_t] = integrate_wave_LF('nx',2001,'nt',2001); cpu_t
>> [xsM,ysM,us,cpu_t] = integrate_wave_LF_vectorized('nx',2001,'nt',2001); cpu_t
>> [xsM,ysM,us,cpu_t] = integrate_wave_LF_vectorized_cols('nx',2001,'nt',2001); cpu_t
```

• As matrices get larger, col approach does better vs for-loops but actually gets worse when matrices are small enough
PDE: Burger’s equation: LF Method, Vectorizing Code, Minimize need for indexing

- function [xsM,tsM,us,cput] =
  integrate_wave_LF_vectorized_cols_min_index_calls(varargin)
  - paras.nx = 101; paras.nt = 101; paras.tf = 0.5;
  - for vac = 1:2:length(varargin)
    - paras.(varargin{vac}) = varargin{vac+1};
  - end
  - paras.dx = 2/(paras.nx-1); paras.dt = paras.tf/(paras.nt-1);
  - xs = linspace(-1,1,paras.nx);
  - u0 = -(1-xs.^2).*xs.*exp(-xs.^2);
  - ts = linspace(0,paras.tf,paras.nt);
  - us = zeros(paras.nt,paras.nx);
  - us(1,:) = u0;
  - ulprev = u0(1:end-2);
  - urprev = u0(3:end);
  - tl = length(ts);
  - tic;
  - us = us';
  - for tc = 2:tl
    - us(2:end-1,tc) = (urprev+ulprev)./2-...
      (paras.dt./paras.dx./2).*...
      (urprev.^2-ulprev.^2);
    - urprev = us(3:end,tc);
    - ulprev = us(1:end-2,tc);
  - end
  - us = us';
  - cput = toc;
  - [xsM,tsM] = meshgrid(xs,ts);
- end
PDE: Comparing using a minimum amount of indexing with other results

• Compare the four codes for a larger number of values:

```matlab
>> [xsM,ysM,us,cpu_t] = integrate_wave_LF('nx',2001,'nt',2001); cpu_t
>> [xsM,ysM,us,cpu_t] = integrate_wave_LF_vectorized('nx',2001,'nt',2001); cpu_t
>> [xsM,ysM,us,cpu_t] = integrate_wave_LF_vectorized_cols('nx',2001,'nt',2001); cpu_t
>> [xsM,ysM,us,cpu_t] = integrate_wave_LF_vectorized_cols_min_index_calls('nx',2001,'nt',2001); cpu_t
```

• We did do better.
PDE: Burger’s equation: Upwinding Method

• Upwinding-Take the flux that is coming into the cell from “upwind” direction

\[ 0 = u_t + uu_x = u_t + \left( u^2 / 2 \right)_x \approx \frac{u_i^{n+1} - u_i^n}{dt} + \frac{(u_{i+1/2}^n)^2 - (u_{i-1/2}^n)^2}{2dx} \]

\[ u_i^{n+1} = u_i^n + \frac{dt}{dx} \frac{(u_{i+1/2}^n)^2 - (u_{i-1/2}^n)^2}{2} \]

\[ u^n_{i+1/2} = \begin{cases} u_i^n & (u_{i+1}^n + u_i^n) / 2 \geq 0 \\ u_{i+1}^n & (u_{i+1}^n + u_i^n) / 2 < 0 \end{cases} \]
PDE: Burger’s equation: Upwinding Method

- **function** [xsM,tsM,us,cput] = integrate_wave_UW(varargin)
- paras.nx = 101; paras.nt = 101; paras.tf = 0.5;
- for vac = 1:2:length(varargin)
  - paras.(varargin{vac}) = varargin{vac+1};
- end
- paras.dx = 2/(paras.nx-1); paras.dt = paras.tf/(paras.nt-1);
- xs = linspace(-1,1,paras.nx);
- u0 = -(1-xs.^2).*xs.*exp(-xs.^2);
- ts = linspace(0,paras.tf,paras.nt);
- us = zeros(paras.nt,paras.nx);
- us(1,:) = u0;
- tic;
- for tc = 2:length(ts)
  - for uc = 2:length(xs)-1
    - up_ave = (us(tc-1,uc+1)+us(tc-1,uc))./2;
    - um_ave = (us(tc-1,uc-1)+us(tc-1,uc))./2;
    - if up_ave >= 0
      - uph = us(tc-1,uc);
    - else
      - uph = us(tc-1,uc+1);
    - end
    - if um_ave >= 0
      - umh = us(tc-1,uc-1);
    - else
      - umh = us(tc-1,uc);
    - end
    - us(tc,uc) = us(tc-1,uc)-
      - (paras.dt/paras.dx).*((uph.^2-umh.^2)./2);
  - end
- end
- cput = toc;
- [xsM,tsM] = meshgrid(xs,ts);
PDE: Check upwind method

• First, try:

```matlab
>> [xsM,tsM,us,cpu_t] = integrate_wave_UW;
plot_pde(xsM,tsM,us)
```

• Note: The amplitude decays less than it did for LF with the same refinement...the miracle of “upwinding”/the problem with using too much “numerical diffusion”

• Second, try:

```matlab
>> [xsM,tsM,us,cpu_t] = integrate_wave_UW('nx',1001,'nt',1001);
plot_pde(xsM,tsM,us)
```
There are three different ways you can index a matrix:

```matlab
>> A = [3,7;2,5;0,-1]
• Use two indices corresponding to row and column
  >> A(1,2)
• Use one index, the “linear” index, corresponding to $n$th entry in the matrix—$A(1) == 3$, $A(2) == 2$ (go down the rows), $A(3) == 0$, $A(4) == 7$ (go to the top of the next column) etc.
  >> A(5)
```
PDE: Vectorizing if statements—Indexing in general

There are three different ways you can index a matrix:

• Use logical indexing. Make a matrix, B, of the same size as A with 0’s and 1’s. You will receive back a vector of values corresponding to the entries equal to 1 in the B matrix:

  >>B = [0 1;1 0; 0 1];

  >>A(B)
PDE: Vectorizing if statements—Indexing with conditional matrices

• Logical indexing is very useful in conjunction with vectorized conditional statements. Consider A. If one wants to know which elements of $A > 1$

  $B = A > 1$

  $A(B)$

• This vectorized conditional matrix, B, has 1’s in it where the condition is true for the components in A and 0’s otherwise

• This allows us to perform vectorized if statements
PDE: Burger’s equation: Upwinding Method with vectorized if statement

- function [xsM,tsM,us,cput] = integrate_wave_UW_vectorized(varargin)
- paras.nx = 101; paras.nt = 101; paras.tf = 0.5;
- for vac = 1:2:length(varargin)
  - paras.(varargin{vac}) = varargin{vac+1};
- end
- paras.dx = 2/(paras.nx-1); paras.dt = paras.tf/(paras.nt-1);
- xs = linspace(-1,1,paras.nx);
- u0 = -(1-xs.^2).*xs.*exp(-xs.^2);
- ts = linspace(0,paras.tf,paras.nt);
- us = zeros(paras.nt,paras.nx);
- us(1,:) = u0;
- ulprev = u0(1:end-1)';
- urprev = u0(2:end)';
- lin_inds = (2:(length(u0)-1))';
- tic;
- us = us';
- for tc = 2:length(ts)
  - uh_aves = (ulprev+urprev)./2;
  - uh_cond = uh_aves < 0;
  - r_inds = lin_inds+uh_cond(2:end);
  - l_inds = lin_inds-1+uh_cond(1:end-1);
  - us(2:end-1,tc) = us(2:end-1,tc-1)-(paras.dt/paras.dx/2).*...
    (us(r_inds,tc-1).^2-us(l_inds,tc-1).^2);
  - ulprev = us(1:end-1,tc);
  - urprev = us(2:end,tc);
- end
- us = us';
- cput = toc;
- [xsM,tsM] = meshgrid(xs,ts);
- end
PDE: Comparing Vectorized Upwinding

• Compare the two codes for small matrices
  \[
  \text{>> } [\text{xsM,tsM,us,cpu}_t] = \text{integrate\_wave\_UW;} \ \text{cpu}_t
  \]
  \[
  \text{>> } [\text{xsM,tsM,us,cpu}_t] = \text{integrate\_wave\_UW\_vectorized;} \ \text{cpu}_t
  \]
• For large matrices
  \[
  \text{>> } [\text{xsM,tsM,us,cpu}_t] = \text{integrate\_wave\_UW('nx',2001,'nt',2001);} \ \text{cpu}_t
  \]
  \[
  \text{>> } [\text{xsM,tsM,us,cpu}_t] = \text{integrate\_wave\_UW\_vectorized('nx',2001,'nt',2001);} \ \text{cpu}_t
  \]
PDE: Vectorizing code

• The larger the objects being processed, the more speedup you’ll receive from vectorizing your code
  – For pdes in 2-d or 3-d spatial dimensions, the benefits of vectorizing code quickly increase
• Vectorizing depends on how you access the components in the matrices (indexing)
  – Row vs col
  – How often you use indices
• You can vectorize if statements
• The codes given above for UW and LF may not be the best implementation for these methods—use profiler, tic, and toc to adjust and optimize the above code
PDE: Burger’s equation: Implicit method

- Consider Burger’s equation in two-dimensions:

\[ 0 = u_t + uu_x \approx \]

\[ \frac{u_i^{n+1} - u_i^n}{dt} + u_i^n \frac{(u_{i+1}^{n+1} - u_{i-1}^{n+1})}{2dx} \]

\[ \left( \frac{dt}{2dx} u_i^n \right) u_{i+1}^{n+1} + u_i^{n+1} + \left( \frac{-dt}{2dx} u_i^n \right) u_{i-1}^{n+1} = u_i^n \]
PDE: Burger’s equation: Implicit method

\[
\begin{bmatrix}
1 & \frac{dt}{2dx}u_1^n & 0 & \cdots & \cdots & 0 \\
-\frac{dt}{2dx}u_2^n & 1 & \frac{dt}{2dx}u_2^n & 0 & \cdots & \cdots \\
0 & -\frac{dt}{2dx}u_3^n & 1 & \frac{dt}{2dx}u_3^n & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \cdots & \ddots \\
0 & \cdots & \cdots & \cdots & \cdots & -\frac{dt}{2dx}u_m^n \\
0 & \cdots & \cdots & \cdots & \cdots & \frac{dt}{2dx}u_m^n
\end{bmatrix}
\begin{bmatrix}
\tilde{u}_{1}^{n+1} \\
\tilde{u}_{2}^{n+1} \\
\tilde{u}_{3}^{n+1} \\
\vdots \\
\vdots \\
\tilde{u}_{m}^{n+1}
\end{bmatrix}
= \begin{bmatrix}
\tilde{u}^n_1 \\
\tilde{u}^n_2 \\
\tilde{u}^n_3 \\
\vdots \\
\vdots \\
\tilde{u}^n_m
\end{bmatrix}

\tilde{u}^{n+1} = A(\tilde{u}^n)^{-1}b(\tilde{u}^n)
PDE: Implicit solve, Vectorized version

```
function [xsM,tsM,us,cput] = integrate_wave_Imp_vectorized(varargin)
    paras.nx = 101; paras.nt = 101; paras.tf = 0.5;
    for vac = 1:2:length(varargin)
        paras.(varargin{vac}) = varargin{vac+1};
    end
    paras.dx = 2/(paras.nx-1); paras.dt = paras.tf/(paras.nt-1);
    xs = linspace(-1,1,paras.nx);
    u0 = -(1-xs.^2).*xs.*exp(-xs.^2);
    ts = linspace(0,paras.tf,paras.nt);
    us = zeros(paras.nt,paras.nx);
    us(1,:) = u0;
    xl = length(xs);
    tl = length(ts);
    tic;
    us = us';
    A = eye(xl-2,xl-2);
    log_inds_l = find(diag(ones(xl-3,1),-1));
    log_inds_r = find(diag(ones(xl-3,1),1));
    for tc = 2:tl
        uprev = us(2:end-1,tc-1);
        A(log_inds_r) = paras.dt./paras.dx./2.*uprev(1:end-1);
        A(log_inds_l) = -paras.dt./paras.dx./2.*uprev(2:end);
        unext = A\uprev;
        us(2:end-1,tc) = unext;
    end
    us = us';
    cput = toc;
    [xsM,tsM] = meshgrid(xs,ts);
end
```
Compare to Explicit

• To compare with explicit, first run the explicit version with the following parameters:

```matlab
>> [xsM,tsM,us,cpu_t] = integrate_wave_LF_vectorized_cols_min_index_calls('nx',10001,'nt',21);
plot_pde(xsM,tsM,us)
```

• Notice the plots become all spiky...bad numerics going on

• Then do the same for the implicit:

```matlab
>> [xsM,tsM,us,cpu_t] = integrate_wave_Imp_vectorized('nx',10001,'nt',21);
plot_pde(xsM,tsM,us)
```

• This time, the plots are fine.

• Advantage of implicit: You can take longer time steps than in explicit methods. Usually useful when diffusion operators come in.

• Disadvantage: Implicit methods much slower per time step.

• Which one you choose depends on desired accuracy and stability
PDE: Implicit solve with Sparse matrices

function [xsM,tsM,us,cput] = integrate_wave_Imp_vectorized_sparse(varargin)
paras.nx = 101; paras.nt = 101; paras.tf = 0.5;
for vac = 1:2:length(varargin)
    paras.(varargin{vac}) = varargin{vac+1};
end
paras.dx = 2/(paras.nx-1); paras.dt = paras.tf/(paras.nt-1);
xs = linspace(-1,1,paras.nx);
u0 = -(1-xs.^2).*xs.*exp(-xs.^2);
ts = linspace(0,paras.tf,paras.nt);
us = zeros(paras.nt,paras.nx);
us(1,:) = u0;
xl = length(xs);
tl = length(ts);
tic;
us = us';
A = speye(xl-2,xl-2);
lin_inds_l = find(diag(ones(xl-3,1),-1));
lin_inds_r = find(diag(ones(xl-3,1),1));
for tc = 2:tl
    uprev = us(2:end-1,tc-1);
    A(lin_inds_r) = paras.dt./paras.dx./2.*uprev(1:end-1);
    A(lin_inds_l) = -paras.dt./paras.dx./2.*uprev(2:end);
    unext = A\uprev;
    us(2:end-1,tc) = unext;
end
us = us';
cput = toc;
[xsM,tsM] = meshgrid(xs,ts);
end
Quick test: No sparse vs sparse

- Check if it works:

  ```matlab
  >> [xsM,tsM,us,cpu_t] = 
     integrate_wave_Imp_vectorized_sparse('nx',101,'nt',101);
  plot_pde(xsM,tsM,us)
  ```

- For default 101 x 101:

  ```matlab
  >> [xsM,tsM,us,cpu_t] = 
     integrate_wave_Imp_vectorized('nt',1001,'nx',1001); cpu_t
  >> [xsM,tsM,us,cpu_t] = 
     integrate_wave_Imp_vectorized_sparse('nt',1001,'nx',1001) ; cpu_t
  ```

- Sparse matrices do not need excess storage for zeros and do not perform excess operations on the zero components of the matrix
function [xsM,tsM,us,cput] = integrate_wave_Imp_vectorized_sparse_bicg(varargin)
paras.nx = 101; paras.nt = 101; paras.tf = 0.5;
for vac = 1:2:length(varargin)
    paras.(varargin{vac}) = varargin{vac+1};
end
paras.dx = 2/(paras.nx-1); paras.dt = paras.tf/(paras.nt-1);
xs = linspace(-1,1,paras.nx);
u0 = -(1-xs.^2).*xs.*exp(-xs.^2);
ts = linspace(0,paras.tf,paras.nt);
us = zeros(paras.nt,paras.nx);
us(1,:) = u0;
xl = length(xs);
tl = length(ts);
tic;
us = us';
A = spye(xl-2,xl-2);
log_inds_l = find(diag(ones(xl-3,1),-1));
log_inds_r = find(diag(ones(xl-3,1),1));
for tc = 2:tl
    uprev = us(2:end-1,tc-1);
    A(log_inds_r) = paras.dt./paras.dx./2.*uprev(1:end-1);
    A(log_inds_l) = -paras.dt./paras.dx./2.*uprev(2:end);
    [unext,flag] = bicg(A,uprev,[],[],[],[],uprev);
    if flag ~= 0
        unext = A\uprev;
    end
    us(2:end-1,tc) = unext;
end
us = us';
cput = toc;
[xsM,tsM] = meshgrid(xs,ts);
Comparisons

• For small matrices, using normal matrices in matlab is fastest but quickly as matrices get larger, using sparse matrices is better (if your matrix has many many zeros and very few nonzero entries)

• For iterative sparse solvers, The “backslash” operator will always win if the matrix is tridiagonal (Matlab treats those special) but if it’s not, as matrices get very large the iterative methods win.

>> [xsM,tsM,us,cpu_t] = integrate_wave_Imp_vectorized_sparse_bicg('nt',1001, 'nx',1001); cpu_t; plot_pde(xsM,tsM,us)
General Matlab thoughts

• Matlab can do some things just as fast as C or Fortran (e.g. matrix multiplication) if not faster (if you haven’t optimally compiled your C or Fortran code)
• Matlab has many built in functions which are near or at C and Fortran efficiency so always ask, "Does Matlab have a function for this?"
• Matlab can also have excessive/unnecessary error checking which can make it slower than other codes (in such cases you might write your own subfunction, e.g. sub2ind)
• Matlab also has basic functions (e.g. if statements and for loops) which are generally slower than in C or Fortran (though they are continually making them smarter/faster)
• Matlab’s Mlint’s suggestions, while often right, can be wrong in particular situations (e.g. “logical indexing may be faster here”)
• Profiling your code occasionally will help avoid any bad programming techniques.