TUNING MATLAB FOR BETTER PERFORMANCE

Kadin Tseng
Boston University
Scientific Computing and Visualization

BOSTON UNIVERSITY

Where to Find Performance Gains ?

- Serial Performance gain
  - Due to memory access
  - Due to caching
  - Due to vector representations
  - Due to compiler
  - Due to other ways
- Parallel performance gain is covered in the MATLAB Parallel Computing Toolbox tutorial

Performance Issues Related to Memory Access

How Does MATLAB Allocate Arrays ?

Each MATLAB array is allocated in contiguous address space.

What happens if you *don't* preallocate array x ?

```
x = 1;
for i=2:4
   x(i) = i;
end
```

To satisfy contiguous memory placement rule, x may need to be moved from one memory segment to another many times during iteration process.

Memory Address	Array Element
1	x(1)
…	. . .
2000	x(1)
2001	x(2)
2002	x(1)
2003	x(2)
2004	x(3)
. . .	. . .
10004	x(1)
10005	x(2)
10006	x(3)
10007	x(4)

Always preallocate array before using it

- Preallocating array to its maximum size prevents all intermediate array movement and copying described.

```
>> A=zeros(n,m); % initialize A to 0
>> A(n,m)=0;      % or touch largest element
```

- If maximum size is not known apriori, estimate with upperbound. Remove unused memory after.

```
>> A=rand(100,100);
>> % . . .
>> % if final size is 60x40, remove unused portion
>> A(61:end,:)=[]; A(:,41:end)=[];  % delete
```

Example

- For efficiency considerations, MATLAB arrays are allocated in contiguous memory space. Arrays follow column-major rule.
- Preallocate array to avoid data movement.

Bad:

```
n=5000;
tic
for i=1:n
    x(i) = i^2;
end
toc
Wallclock time = 0.00046 seconds
```

not_allocate.m

Good:

```
n=5000; x = zeros(n, 1);
tic
for i=1:n
    x(i) = i^2;
end
toc
Wallclock time = 0.00004 seconds
```

allocate.m

The timing data are recorded on older cluster. The actual times on your computer may vary depending on the processor.

Lazy Copy

MATLAB uses pass-by-reference if passed array is used without changes; a copy will be made if the array is modified. MATLAB calls it “lazy copy.” Example:

```
function y = lazyCopy(A, x, b, change)
If change, A(2,3) = 23; end        % forces a local copy of a
y = A*x + b;     % use x and b directly from calling program
pause(2)         % keep memory longer  to see it  in Task Manager
```

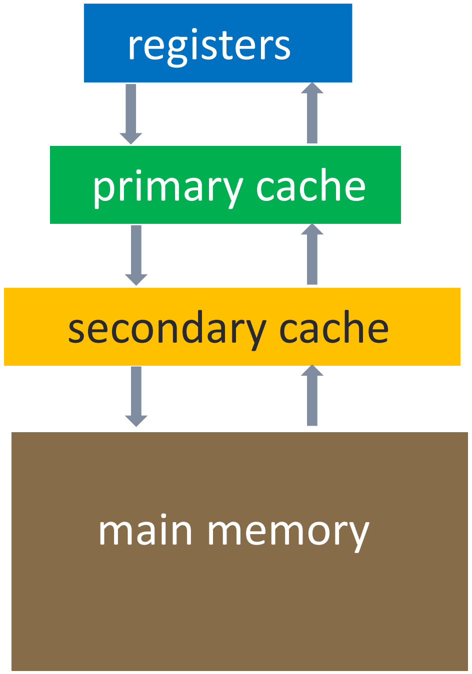
On Windows, use Task Manager to monitor memory allocation history.

```
>> n = 5000; A = rand(n); x = rand(n,1); b = rand(n,1);
>> y = lazyCopy(A, x, b, 0);       % no copy; pass by reference
>> y = lazyCopy(A, x, b, 1);       % copy; pass by value
```

Performance Issues Related to Caching

Cache

- Cache is a small chunk of fast memory between the main memory and the registers



Cache (2)

- If variables are fetched from cache, code will run faster since cache memory is much faster than main memory
- Variables are moved from main memory to cache stages

Cache (3)

- Why not just make the main memory out of the same stuff as cache?
  - Expensive
  - Runs hot
  - This was actually done in Cray computers
    - Liquid cooling system
  - Currently, special clusters (on XSEDE.org) available with very substantial flash main memory for I/O-bound applications

Cache (4)

- Cache hit
  - Required variable is in cache
- Cache miss
  - Required variable not in cache
  - If cache is full, something in there must be thrown out (sent back to main memory) to make room
  - Want to minimize number of cache misses

Cache (5)

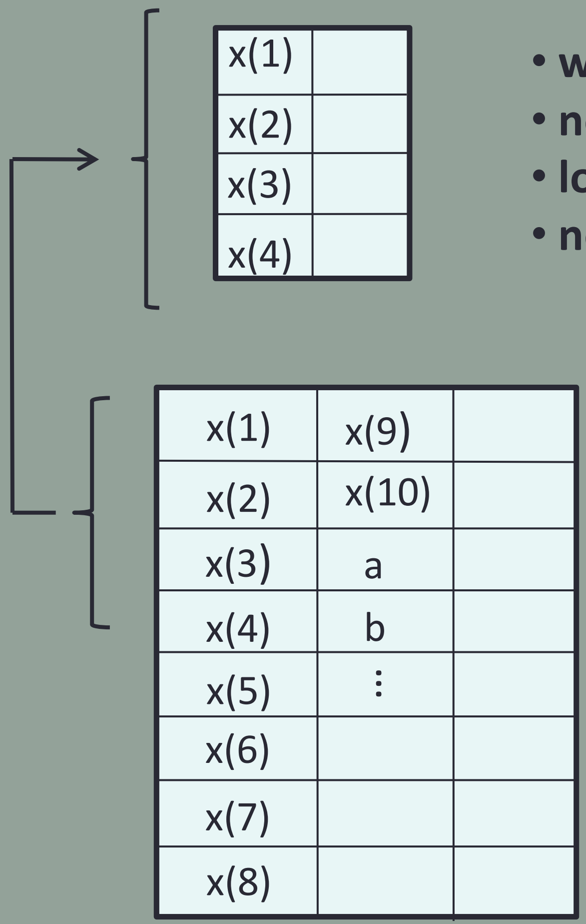
"mini" cache
holds 2 lines, 4 words each

x(1)	x(9)	
x(2)	x(10)	
x(3)	a	
x(4)	b	
x(5)	⋮	
x(6)		
x(7)		
x(8)		

```
for i=1:10
    x(i) = i;
end
```

Main memory

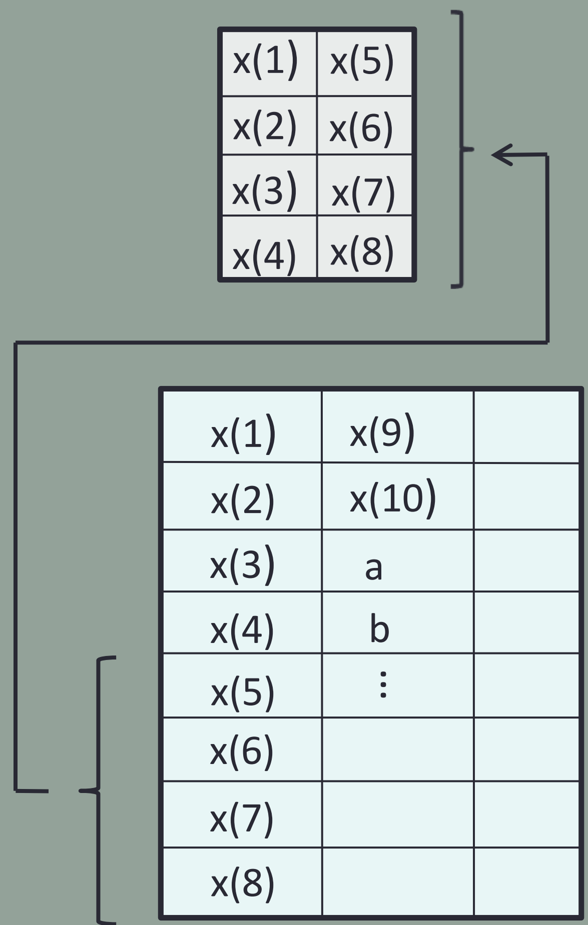
Cache (6)



- will ignore i for simplicity
- need x(1), not in cache → cache miss
- load line from memory into cache
- next 3 loop indices result in cache hits

```
for i=1:10
   x(i) = i;
end
```

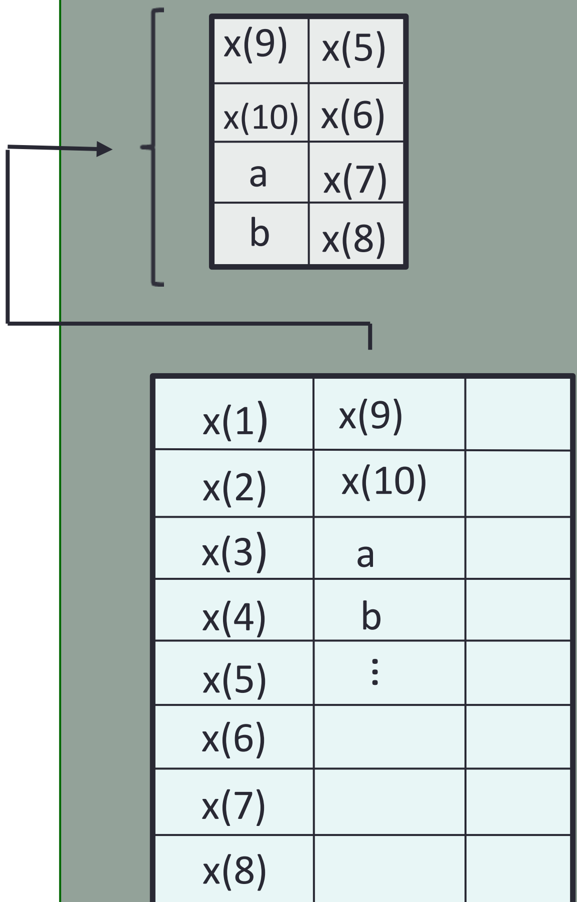
Cache (7)



- need x(5), not in cache → cache miss
- load line from memory into cache
- free ride next 3 loop indices → cache hits

```
for i=1:10
    x(i) = i;
end
```

Cache (8)



- **need x(9), not in cache --> cache miss**
- **load line from memory into cache**
- **no room in cache!**
- **replace old line**

```
for i=1:10
    x(i) = i;
end
```

Cache (9)

- Multidimensional array is stored in column-major order:

  x(1,1)

  x(2,1)

  x(3,1)

  .

  .

  x(1,2)

  x(2,2)

  x(3,2)

  .

  .

For-loop Order

- Best if inner-most loop is for array left-most index, etc. (column-major)

Bad:

```
n=5000; x = zeros(n);
for i = 1:n       % rows
   for j = 1:n    % columns
     x(i,j) = i+(j-1)*n;
   end
end

Wallclock time = 0.88 seconds
```

forij.m

Good:

```
n=5000; x = zeros(n);
for j = 1:n       % columns
   for i = 1:n    % rows
     x(i,j) = i+(j-1)*n;
   end
end

Wallclock time = 0.48 seconds
```

forji.m

- For a multi-dimensional array, *x(i,j)*, the 1D representation of the same array, x(k), follows column-wise order and inherently possesses the contiguous property

Compute In-place

- Compute and save array in-place improves performance and reduces memory usage

Bad:

```
x = rand(5000);
tic
y = x.^2;
toc

Wallclock time = 0.30 seconds
```

not_inplace.m

Good:

```
x = rand(5000);
tic
x = x.^2;
toc

Wallclock time = 0.11 seconds
```

inplace.m

Caveat: May not be worthwhile if it involves data type or size changes …

Eliminate redundant operations in loops

Bad:

```
for i=1:N
    x = 10;
        .
        .
end
```

Good:

```
x = 10;
for i=1:N
        .
        .
end
```

Better performance to use vector than loops

Loop Fusion

Bad:

```
for i=1:N
    x(i) = i;
end
for i=1:N
    y(i) = rand();
end
```

Good:

```
for i=1:N
    x(i) = i;
    y(i) = rand();
end
```

- Reduces for-loop overhead
- More important, improve chances of pipelining
- Loop fisssion splits statements into multiple loops

Avoid *if* statements within loops

Bad:

if has overhead cost and may inhibit pipelining

```
for i=1:N
    if i == 1
        %perform i=1 calculations
    else
        %perform i>1 calculations
    end
end
```

Good:

```
%perform i=1 calculations
for i=2:N
        %perform i>1 calculations
end
```

Divide is more expensive than multiply

- Intel x86 clock cycles per operation
  - add 3-6
  - multiply 4-8
  - divide 32-45
- Bad:

```
c = 4;
for i=1:N
   x(i)=y(i)/c;
end
```

- Good:

```
s = 1/c;
for i=1:N
   x(i) = y(i)*s;
end
```

Function Call Overhead

Bad:

```
for i=1:N
    myfunc(i);
end
```

```
function myfunc(i)
  do stuff
end
```

Good:

```
myfunc2(N);
```

→

```
function myfunc2(N)
    for i=1:N
        do stuff
    end
end
```

Function m-file is precompiled to lower overhead for repeated usage. Still, there is an overhead . Balance between modularity and performance.

Minimize calls to math & arithmetic operations

Bad:

```
for i=1:N
   z(i) = log(x(i)) + log(y(i));
   v(i) = x(i) + x(i)^2 + x(i)^3;
end
```

Good:

```
for i=1:N
    z(i) = log(x(i) * y(i));
    v(i) = x(i)*(1+x(i)*(1+x(i)));
end
```

Special Functions for Real Numbers

MATLAB provides a few functions for processing *real* number specifically. These functions are more efficient than their generic versions:

- realpow – power for real numbers
- realsqrt – square root for real numbers
- reallog – logarithm for real numbers
- realmin/realmax – min/max for real numbers

```
n = 1000; x = 1:n;
x = x.^2;
tic
x = sqrt(x);
toc

Wallclock time =  0.00022 seconds
```

square_root.m

```
n = 1000; x = 1:n;
x = x.^2;
tic
x = realsqrt(x);
toc

Wallclock time = 0.00004 seconds
```

real_square_root.m

- *isreal* reports whether the array is real
- *single/double* converts data to single-, or double-precision

Vector Is Better Than Loops

- MATLAB is designed for vector and matrix operations. The use of *for*-loop, in general, can be expensive, especially if the loop count is large and nested.
- Without array pre-allocation, its size extension in a for-loop is costly as shown before.
- When possible, use vector representation instead of *for*-loops.

```
i = 0;
for t = 0:.01:100
    i = i + 1;
    y(i) = sin(t);
end

Wallclock time = 0.1069 seconds
```

for_sine.m

```
t = 0:.01:100;
y = sin(t);

Wallclock time = 0.0007 seconds
```

vec_sine.m

Vector Operations of Arrays

```
>> A = magic(3) % define a 3x3 matrix A
A =
     8     1     6
     3     5     7
     4     9     2
>> B = A^2;          % B = A * A;
>> C =  A + B;
>> b = 1:3           % define b as a 1x3 row vector
b =
     1     2     3
>> [A, b']           % add b transpose as a 4th column to A
ans =
     8     1     6     1
     3     5     7     2
     4     9     2     3
```

Vector Operations

```
>> [A; b]                % add b as a 4th row to A
ans =
     8     1     6
     3     5     7
     4     9     2
     1     2     3
>> A = zeros(3)          % zeros generates 3 x 3 array of 0's
A =
     0     0     0
     0     0     0
     0     0     0
>> B = 2*ones(2,3)   % ones generates 2 x 3 array of 1's
B =
     2     2     2
     2     2     2
```

Alternatively,

```
>> B = repmat(2,2,3)    % matrix replication
```

Vector Operations

```
>> y = (1:5)';
>> n = 3;
>> B = y(:, ones(1,n))      % B = y(:, [1 1 1]) or B=[y y y]
B =
     1     1     1
     2     2     2
     3     3     3
     4     4     4
     5     5     5
```

Again, *B* can be generated via repmat as

```
>> B = repmat(y, 1, 3);
```

Vector Operations

```
>> A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
>> B = A(:, [1 3 2]) % switch 2nd and third columns of A
B =
     8     6     1
     3     7     5
     4     2     9
>> A(:, 2) = [  ] % delete second column of A
A =
     8     6
     3     7
     4     2
```

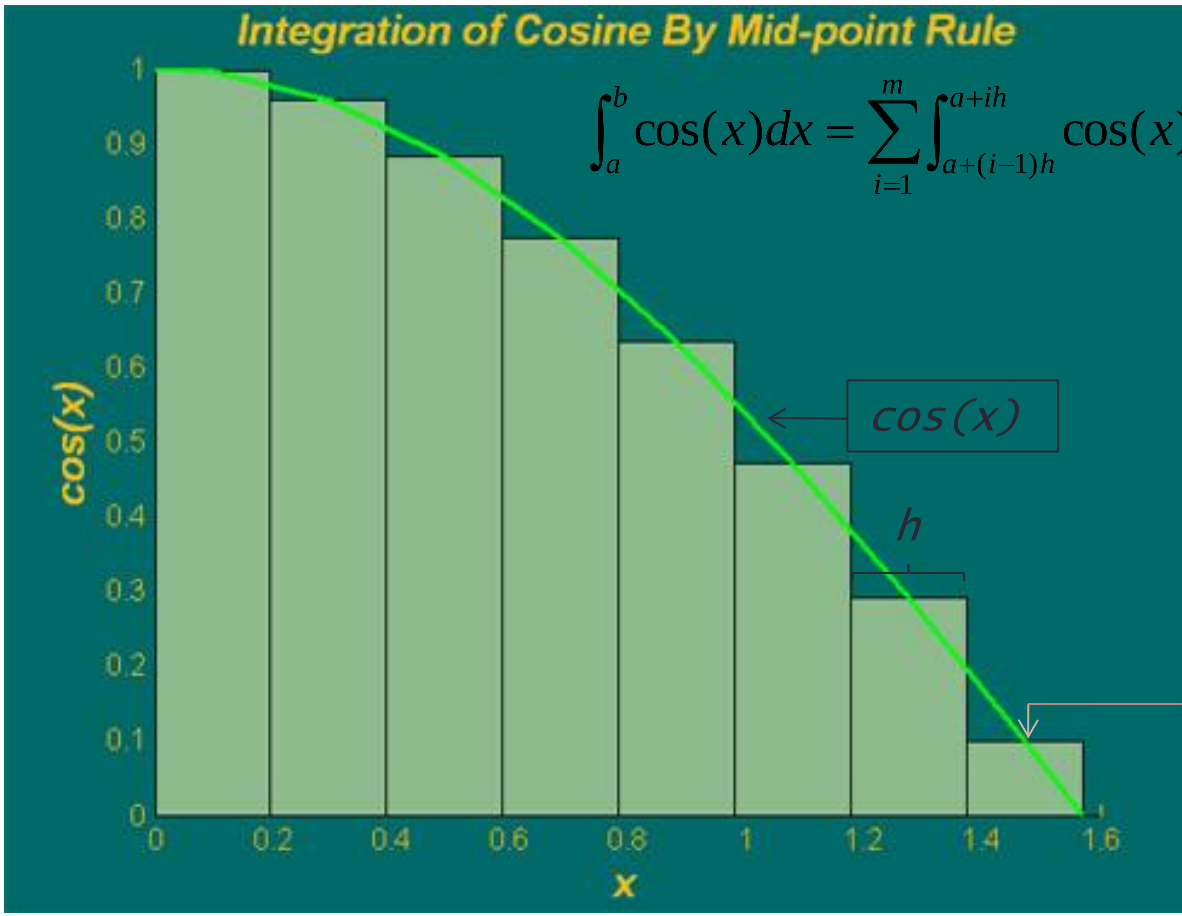
Vector Utility Functions

Function	Description
all	Test to see if all elements are of a prescribed value
any	Test to see if any element is of a prescribed value
zeros	Create array of zeroes
ones	Create array of ones
repmat	Replicate and tile an array
find	Find indices and values of nonzero elements
diff	Find differences and approximate derivatives
squeeze	Remove singleton dimensions from an array
prod	Find product of array elements
sum	Find the sum of array elements
cumsum	Find cumulative sum
shiftdim	Shift array dimensions
logical	Convert numeric values to logical
sort	Sort array elements in ascending /descending order

Integration Example

- Integral is area under *cosine* function in range of 0 to $\pi/2$
- Equals to sum of all rectangles (width times height of bars)

$$\int_a^b \cos(x)dx = \sum_{i=1}^{m} \int_{a+(i-1)h}^{a+ih} \cos(x)dx \approx \sum_{i=1}^{m} \cos(a+(i-\tfrac{1}{2})h)h$$

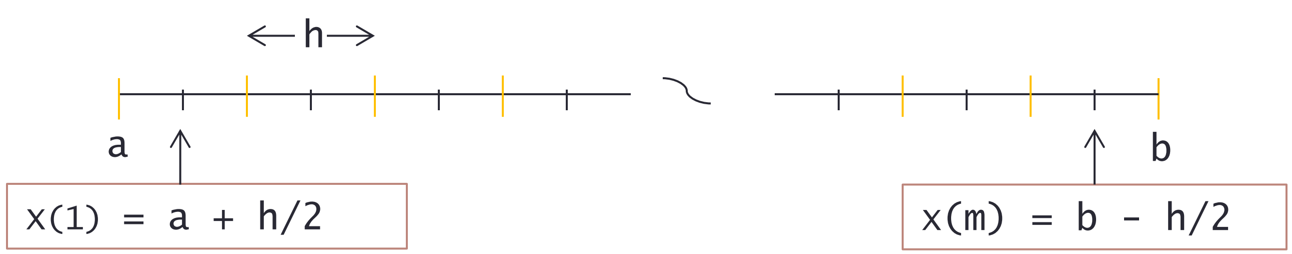



```
a = 0; b = pi/2;  % range
m = 8;  % # of increments
h = (b-a)/m;  % increment
```

mid-point of increment

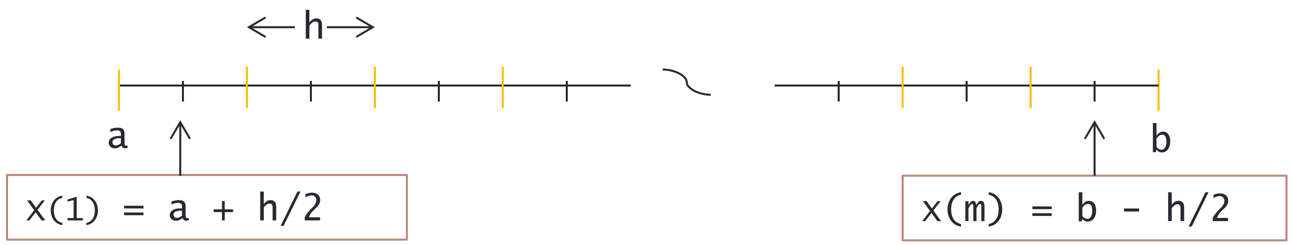
Integration Example — using for-loop

```
% integration with for-loop
tic
   m = 100;
   a = 0;                  % lower limit of integration
   b = pi/2;               % upper limit of integration
   h = (b - a)/m;          % increment length
   integral = 0;           % initialize integral
   for i=1:m
     x = a+(i-0.5)*h;      % mid-point of increment i
     integral = integral + cos(x)*h;
   end
toc
```



Integration Example — using vector form

```
% integration with vector form
tic
   m = 100;
   a = 0;                 % lower limit of integration
   b = pi/2;              % upper limit of integration
   h = (b - a)/m;         % increment length
   x = a+h/2:h:b-h/2;     % mid-point of m increments
   integral = sum(cos(x))*h;
toc
```



Integration Example Benchmarks

increment m	for-loop	Vector
10000	0.00044	0.00017
20000	0.00087	0.00032
40000	0.00176	0.00064
80000	0.00346	0.00130
160000	0.00712	0.00322
320000	0.01434	0.00663

- Timings (seconds) obtained on Intel Core i5 3.2 GHz PC
- Computational effort linearly proportional to # of increments.

Laplace Equation (Steady incompressible potential flow)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Boundary Conditions:

$$u(x,0) = \sin(\pi x) \qquad 0 \le x \le 1$$

$$u(x,1) = \sin(\pi x)e^{-\pi} \qquad 0 \le x \le 1$$

$$u(0,y) = u(1,y) = 0 \qquad 0 \le y \le 1$$

Analytical solution:

$$u(x,y) = \sin(\pi x)e^{-\pi y} \qquad 0 \le x \le 1; \;\; 0 \le y \le 1$$

Finite Difference Numerical Discretization

Discretize equation by centered-difference yields:

$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4} \qquad i = 1,2,\ldots,m; \;\; j = 1,2,\ldots,m$$

where *n* and *n+1* denote the current and the next time step, respectively, while

$$u_{i,j}^{n} = u^{n}(x_i, y_j) \qquad i = 0,1,2,\ldots,m+1; \;\; j = 0,1,2,\ldots,m+1$$

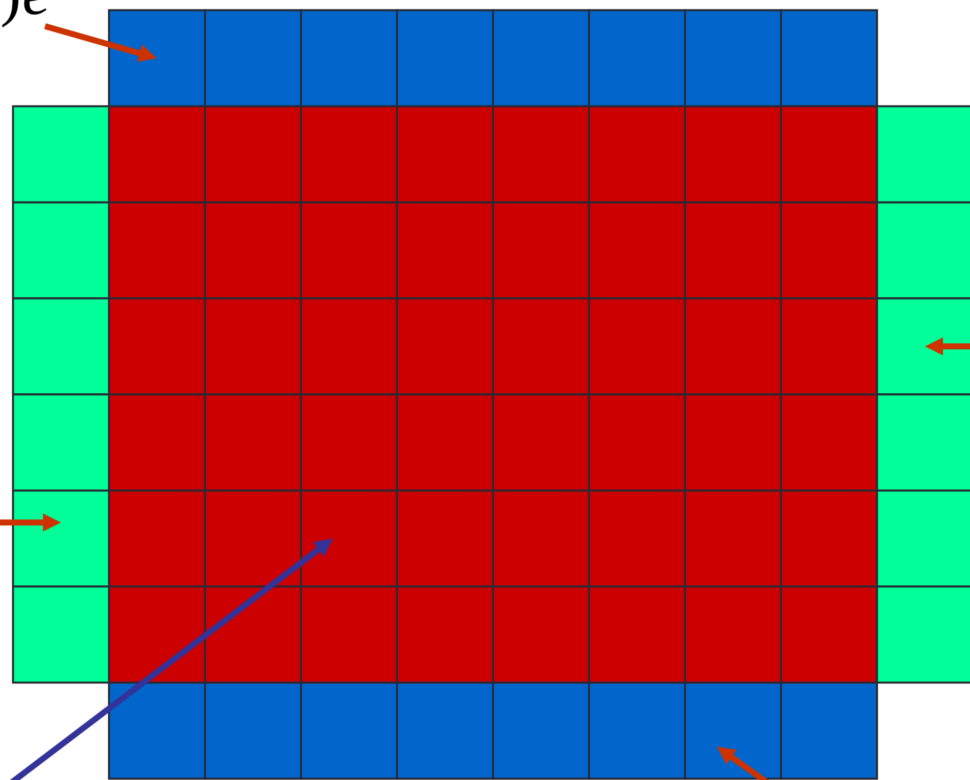
$$= u^{n}(i\Delta x, j\Delta y)$$

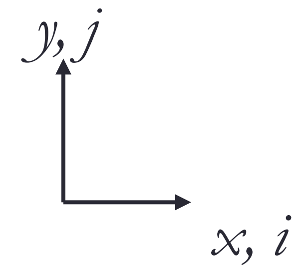
For simplicity, we take

$$\Delta x = \Delta y = \frac{1}{m+1}$$

Computational Domain

$$u(x,1) = \sin(\pi x)e^{-\pi}$$





$$u(1,y) = 0$$

$$u(0,y) = 0$$

$$u(x,0) = \sin(\pi x)$$

$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4}$$

$$i = 1,2,\ldots,m; \quad j = 1,2,\ldots,m$$

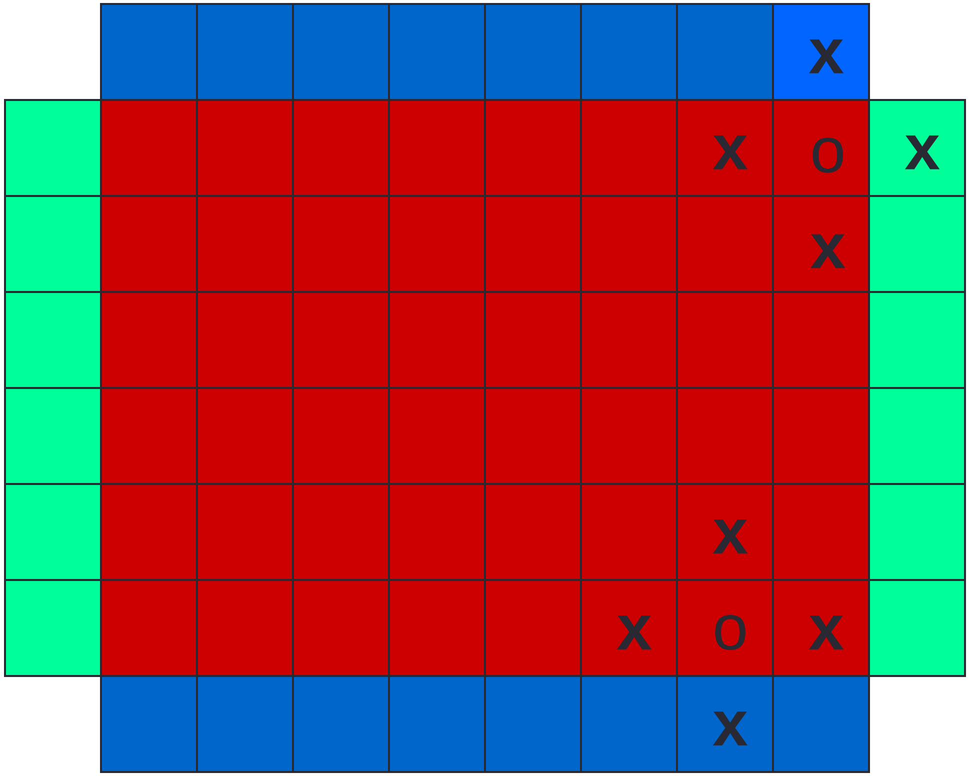
Five-point Finite-difference Stencil

Interior cells.

Where solution of the Laplace equation is sought.

Exterior cells.

Green cells denote cells where homogeneous boundary conditions are imposed while non-homogeneous boundary conditions are colored in blue.



SOR Update Function

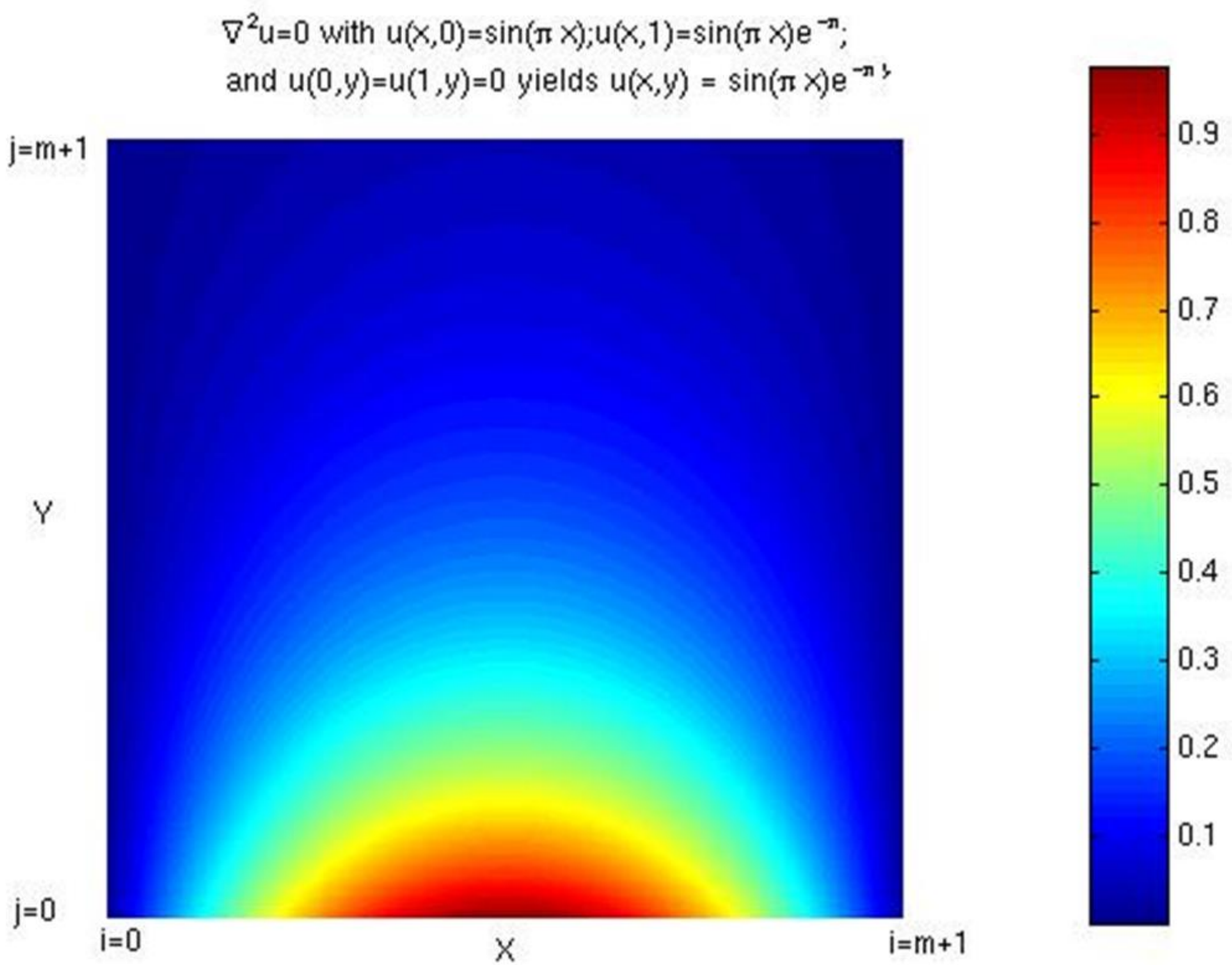
How to vectorize it ?

1. Remove the *for-loops*
2. Define i = ib:2:ie;
3. Define j = jb:2:je;
4. Use *sum* for del

```
% original code fragment
jb = 2; je = n+1; ib = 3; ie = m+1;
for  i=ib:2:ie
   for  j=jb:2:je
        up = ( u(i   ,j+1) + u(i+1,j  ) + ...
                  u(i-1,j    ) + u(i    ,j-1) )*0.25;
        u(i,j) = (1.0 - omega)*u(i,j) +omega*up;
        del = del + abs(up-u(i,j));
   end
end
```

```
% equivalent vector code fragment
jb = 2; je = n+1; ib = 3; ie = m+1;
i = ib:2:ie;  j = jb:2:je;
up = ( u(i   ,j+1) + u(i+1,j   ) + ...
          u(i-1,j    ) + u(i    ,j-1) )*0.25;
u(i,j) = (1.0 - omega)*u(i,j) + omega*up;
del = sum(sum(abs(up-u(i,j))));
```

Solution Contour Plot



SOR Timing Benchmarks

m Matrix size	Wallclock ssor2Dij for loops	Wallclock ssor2Dji reverse loops	Wallclock ssor2Dv vector
128	1.01	0.98	0.26
256	8.07	7.64	1.60
512	65.81	60.49	11.27
1024	594.91	495.92	189.05

Summation

- For global sum of 2D matrices: *sum(sum(A)) or sum(A(:))*

  Example: *which is more efficient ?*

  ```
  A = rand(1000);
  tic,sum(sum(A)),toc
  tic,sum(A(:)),toc
  ```

  No appreciable performance difference; latter more compact.

  Your application calls for summing a matrix along rows (dim=2) multiple times (inside a loop). Example:

  ```
  A = rand(1000);
  tic, for t=1:100,sum(A,2);end, toc
  ```

- MATLAB matrix memory ordering is by column. Better performance if sum by column. Swap the two indices of *A* at the outset.

  Example: B=A'; tic, for t=1:100, sum(B,1);end, toc *(See twosums.m)*

Logical Array helpful for Vectorization

- Sometimes, logical array is used to retain target array's shape

Scalar example:

```
a = rand(4,3);
b =rand(size(a));
c = zeros(size(b));
b(1,3) = 0; b(3,2) = 0;
for j=1:3
  for i=1,4
    if (b(i,j) ~= 0) then
       c(i,j) = a(i,j)/b(i,j);
    end
  end
end
```

Vector example:

```
% e is true (1) for all b not = 0
e = b~=0
 e =
     1    1    0
     1    1    1
     1    0    1
     1    1    1
c(e) = a(e)./b(e)    %  c = 0 ∀ b
= 0
c =
    0.9768   1.4940   0
    2.3896   0.4487   0.0943
    0.7821   0           0.2180
   11.3867   0.0400   1.2741
```

Other Tips

- Generally better to use function rather than script
  - Script m-file is loaded into memory and evaluate one line at a time. Subsequent uses require reloading.
  - Function m-file is compiled into a pseudo-code and is loaded on first application. Subsequent uses of the function will be faster without reloading.
  - Function is modular; self cleaning; reusable.
- Global variables are expensive; difficult to track.
- Don’t reassign array that results in change of data type or shape
- Limit m-files size and complexity
- Structure of arrays more memory-efficient than array of structures

Memory Management

- Maximize memory availability.
  - 32-bit systems < 2 or 3 GB
  - 64-bit systems running 32-bit MATLAB < 4GB
  - 64-bit systems running 64-bit MATLAB < 8TB

    (96 GB on some Katana nodes)
- Minimize memory usage. (Details to follow …)

Minimize Memory Usage

- Use *clear, pack* or other memory saving means when possible. If double precision (default) is not required, the use of 'single' data type could save substantial amount of memory. For example,

  >> x=ones(10,'single'); y=x+1; % y inherits single from x

- Use *sparse* to reduce memory footprint on sparse matrices

  >> n=3000; A = zeros(n); A(3,2) = 1; B = ones(n);

  *>> tic, C = A*B; toc % 6 secs*

  >> As = sparse(A);

  *>> tic, D = As*B; toc % 0.12 secs; D not sparse*

- Be aware that array of structures uses more memory than structure of arrays. (pre-allocation is good practice too for structs!)

Minimize Memory Uage

- For batch jobs, use “matlab –nojvm …” saves lots of memory
- Memory usage query

  For Linux:

  scc1% top

  For Windows:

  >> m = feature('memstats');   % largest contiguous free block

  Use MS Windows Task Manager to monitor memory allocation.
- On multiprocessor systems, distribute memory among processors

Compilers

- *mcc* is a MATLAB compiler:
  - It compiles m-files into C codes, object libraries, or stand-alone executables.
  - A stand-alone executable generated with **mcc** can run on *compatible platforms* without an installed MATLAB or a MATLAB license.
  - On special occasions, MATLAB access may be denied if all licenses are checked out. Running a stand-alone requires NO licenses and no waiting.
  - It is not meant to facilitate any performance gains.
- *coder* — m-file to C code converter

mcc *example*

How to build a standalone executable on Windows

>> mcc –o twosums –m twosums

How to run executable on Windows' Command Promp (dos)

Command prompt:> twosums 3000 2000

Details:

- twosums.m is a function m-file with 2 input arguments
- Input arguments to code are processed as strings by *mcc*. Convert with *str2double*: *if isdeployed, N=str2double(N); end*
- Output cannot be returned; either save to file or display on screen.
- The executable is twosums.exe

MATLAB Programming Tools

- profile - profiler to identify "hot spots" for performance enhancement.
- mlint - for inconsistencies and suspicious constructs in m-files.
- debug - MATLAB debugger.
- guide - Graphical User Interface design tool.

MATLAB Profiler

To use profile viewer, DONOT start MATLAB with –nojvm option

```
>> profile on –detail 'builtin' –timer 'real'
>> serial_integration2 % run code to be profiled
>> profile viewer    % view profiling data
>> profile off         % turn off profiler
```

Turn on profiler. Time reported in wall clock. Include timings for built-in functions.

How to Save Profiling Data

Two ways to save profiling data:

1. Save into a directory of HTML files
   Viewing is static, i.e., the profiling data displayed correspond to a prescribed set of options. View with a browser.
2. Saved as a MAT file
   Viewing is dynamic; you can change the options. Must be viewed in the MATLAB environment.

Profiling – save as HTML files

Viewing is static, *i.e.,* the profiling data displayed correspond to a prescribed set of options. View with a browser.

```
>> profile on
>> serial_integration2
>> profile viewer
>> p = profile('info');
>> profsave(p, ‘my_profile') % html files in my_profile dir
```

Profiling – save as MAT file

Viewing is dynamic; you can change the options. Must be viewed in the MATLAB environment.

```
>> profile on
>> serial_integration2
>> profile viewer
>> p = profile('info');
>> save myprofiledata p
>> clear p
>> load myprofiledata
>> profview(0,p)
```

MATLAB Editor

MATLAB editor does a lot more than file creation and editing …

- Code syntax checking
- Code performance suggestions
- Runtime code debugging

Running MATLAB

- scc1% matlab -nodisplay –nosplash –r “n=4, myfile(n); exit”
- *Add –nojvm to save memory if Java is not required*
- For batch jobs on the SCC, put above command in a batch script
- Visit http://www.bu.edu/tech/about/research/training/scv-software-packages/matlab/matlab-batch for instructions on how to run MATLAB batch jobs.

Multiprocessing with MATLAB

- Explicit parallel operations

  MATLAB Parallel Computing Toolbox Tutorial

  www.bu.edu/tech/research/training/tutorials/matlab-pct/

- Implicit parallel operations
  - Require shared-memory computer architecture (*i.e.,* multicore).
  - Feature on by default. Turn it off with

    scc1% matlab –singleCompThread
  - Specify number of threads with maxNumCompThreads

    (deprecated in future).
  - Activated by vector operation of applications such as hyperbolic or trigonometric functions, some LaPACK routines, Level-3 BLAS.
  - See “Implicit Parallelism” section of the above link.

Where Can I Run MATLAB ?

- There are a number of ways:
- Buy your own student version.
- http://www.bu.edu/tech/desktop/site-licensed-software/mathsci/matlab/faqs/#student
- Check your own department to see if there is a computer
- lab with installed MATLAB
- With a valid BU userid, the engineering grid will let you gain
- access remotely.
- http://collaborate.bu.edu/moin/GridInstructions
- If you have a Mac, Windows PC or laptop, you may have to
- sync it with Active Directory (AD) first:
- http://www.bu.edu/tech/accounts/remote/away/ad/
- acs-linux.bu.edu, scc1.bu.edu
- http://www.bu.edu/tech/desktop/site-licensed-
- software/mathsci/mathematica/student-resources-at-bu

Useful SCV Info

- **SCV home page** (www.bu.edu/tech/research)
- **Resource Applications**
  www.bu.edu/tech/accounts/special/research/accounts
- **Help**
  - **System**
    - **help@scc.bu.edu, bu.service-now.com**
  - Web-based tutorials
    (www.bu.edu/tech/research/training/tutorials)
    (MPI, OpenMP, MATLAB, IDL, Graphics tools)
  - HPC consultations by appointment
    - Yann Tambouret (yannpaul@bu.edu)
    - Katia Oleinik (koleinik@bu.edu)
    - Kadin Tseng (kadin@bu.edu)