Writing Fast MATLAB® Code

Pascal Getreuer, February 2009

Contents

1 Introduction

MATLAB is a *prototyping* environment, meaning it focuses on the ease of development with language flexibility, interactive debugging, and other conveniences lacking in performance-oriented languages like C and FORTRAN. While MATLAB may not be as fast as C, there are ways to bring it closer.

> **Disclaimer** This is not a beginner's tutorial to MATLAB, but a tutorial on performance. The methods discussed here are generally fast, but *no claim is made* on what is fastest. Use at your own risk.

Before beginning our main topic, let's step back for a moment and ask ourselves what we really want. Do we want the fastest possible code?—if so, we should switch to a performance language like C. Probably more accurately, we want to spend less time *total* from developing, debugging, running, and until finally obtaining results. This section gives some tips on working smart, not hard.

M-Lint Messages

In more recent versions of MATLAB, the integrated editor automatically gives feedback on possible code errors and suggested optimizations. These messages are helpful in improving a program and in learning more about MATLAB.

```
for k = 1:NumTrials
    r = rand;
    x(k) = runsim(r);
end   'x' might be growing inside a loop. Consider preallocating for speed.

hist(x);  % Plot histogram of simulation outputs
```

Hold the mouse cursor over underlined code to read a message. Or, see all M-Lint messages at once with Tools → M-Lint → Show M-Lint Report.

Organization

- **Use a separate folder for each project.**

  A separate folder for each project keeps related things together and simplifies making copies and backups of project files.

- **Write header comments, especially H1.**

  The first line of the header comment is called the H1 comment. Type `help(cd)` to get a listing of all project files and their H1 comments.

- **Save frequent console commands as a script.**

  If you find yourself repeating certain commands on the console, save them as a script. Less typing means fewer opportunities for typos.

Avoid losing data

- **Don't use `clear all` in a script.**

  This is an unfortunate common practice—any important variables in the base workspace will be irretrievably lost.

- **Beware of clobber.**

  "File clobber" refers to the kind of data loss when a file is accidentally overwritten with another one having the same filename. This phenomenon can occur with variables as well:

```
>> result = big_operation(input1);  % Store the result of a time-consuming operation
...
>> result = big_operation(input2);  % Run again with another input
```

  Variable `result` was clobbered and the first output was lost.

- **Beware of what can crash MATLAB.**

  While MATLAB is generally reliable, crashes are possible when using third-party MEX functions or extremely memory-intensive operations, for example, with video and very large arrays.

Now with good working habits covered, we begin our discussion of writing fast MATLAB code. The rest of this article is organized by topic, first on techniques that are useful in general application, and next on specific computational topics (table of contents is on the first page).

The Profiler

MATLAB 5.0 (R10) and newer versions include a tool called the profiler that helps identify bottlenecks in a program. Use it with the `profile` command:

`profile on`	Turn the profiler on
`profile off`	Turn it back off
`profile clear`	Clear profile statistics
`profile report`	View the results from the profiler

For example, consider profiling the following function:

```
function result = example1(Count)

for k = 1:Count
   result(k) = sin(k/50);

   if result(k) < -0.9
      result(k) = gammaln(k);
   end
end
```

To analyze the efficiency this function, first enable and clear the profiler, run the function, and then view the profile report:

```
>> profile on, profile clear
>> example1(5000);
>> profile report
```

There is a slight parsing overhead when running code for the first time; run the test code twice and time the second run. The `profiler report` command shows a report. Depending on the system, profiler results may differ from this example.

MATLAB Profile Report: Summary

Report generated 30-Jul-2004 16:57:01

Total recorded time:	3.09 s
Number of M-functions:	4
Clock precision:	0.016 s

Function List

Name	Time		Time	Time/call	Self time		Location
example1	3.09	100.0%	1	3.094000	2.36	76.3%	~/example1.m
gammaln	0.73	23.7%	3562	0.000206	0.73	23.7%	../toolbox/matlab/specfun/gammaln.m
profile	0.00	0.0%	1	0.000000	0.00	0.0%	../toolbox/matlab/general/profile.m
profreport	0.00	0.0%	1	0.000000	0.00	0.0%	../toolbox/matlab/general/profreport.m

Clicking the "example1" link gives more details:

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time
4	`result(k) = sin(k/50);`	5000	2.11 s	68%
7	`result(k) = gammaln(k);`	721	0.84 s	27%
6	`if result(k) < -0.9`	5000	0.14 s	5%
Totals			3.09 s	100%

The most time-consuming lines are displayed, along with time, time percentage, and line number. The most costly lines are the computations on lines 4 and 7.

Another helpful section of the profile report is "M-Lint Results," which gives feedback from the M-Lint code analyzer. Possible errors and suggestions are listed here.

M-Lint results

Line number	Message
4	'`result`' might be growing inside a loop. Consider preallocating for speed.
7	'`result`' might be growing inside a loop. Consider preallocating for speed.

(Preallocation is discussed in the next section.)

The profiler has limited time resolution, so to profile a piece of code that runs too quickly, run the test code multiple times with a loop. Adjust the number of loop iterations so that the time it takes to run is noticeable. More iterations yields better time resolution.

The profiler is an essential tool for identifying bottlenecks and per-statement analysis, however, for more accurate timing of a piece of code, use the `tic`/`toc` stopwatch timer.

```
>> tic; example1(5000); toc;
Elapsed time is 3.082055 seconds
```

For serious benchmarking, also close your web browser, anti-virus, and other background processes that may be taking CPU cycles.

Array Preallocation

MATLAB's matrix variables have the ability to dynamically augment rows and columns. For example,

```
>> a = 2

a =
     2

>> a(2,6) = 1

a =
     2     0     0     0     0     0
     0     0     0     0     0     1
```

MATLAB automatically resizes the matrix. Internally, the matrix data memory must be reallocated with larger size. If a matrix is resized repeatedly—like within a loop—this overhead can be significant. To avoid frequent reallocations, *preallocate* the matrix with the `zeros` command.
Consider the code:

```
a(1) = 1;
b(1) = 0;

for k = 2:8000
   a(k) = 0.99803 * a(k - 1) - 0.06279 * b(k - 1);
   b(k) = 0.06279 * a(k - 1) + 0.99803 * b(k - 1);
end
```

This code takes 0.47 seconds to run. After the `for` loop, both arrays are row vectors of length 8000, thus to preallocate, create empty `a` and `b` row vectors each with 8000 elements.

```
a = zeros(1,8000);     % Preallocation
b = zeros(1,8000);
a(1) = 1;
b(1) = 0;

for k = 2:8000
   a(k) = 0.99803 * a(k - 1) - 0.06279 * b(k - 1);
   b(k) = 0.06279 * a(k - 1) + 0.99803 * b(k - 1);
end
```

With this modification, the code takes only 0.14 seconds (over three times faster). Preallocation is often easy to do, in this case it was only necessary to determine the right preallocation size and add two lines.

What if the final array size can vary? Here is an example:

```
a = zeros(1,10000);      % Preallocate
count = 0;

for k = 1:10000
   v = exp(rand*rand);

   if v > 0.5            % Conditionally add to array
      count = count + 1;
      a(count) = v;
   end
end

a = a(1:count);          % Trim extra zeros from the results
```

The average run time of this program is 0.42 seconds without preallocation and 0.18 seconds with it.

Preallocation is also beneficial for cell arrays, using the `cell` command to create a cell array of the desired size.

JIT Acceleration

MATLAB 6.5 (R13) and later feature the Just-In-Time (JIT) Accelerator for improving the speed of M-functions, particularly with loops. By knowing a few things about the accelerator, you can improve its performance.

The JIT Accelerator is enabled by default. To disable it, type "`feature accel off`" in the console, and "`feature accel on`" to enable it again.

As of MATLAB R2008b, only a subset of the MATLAB language is supported for acceleration. Upon encountering an unsupported feature, acceleration processing falls back to non-accelerated evaluation. Acceleration is most effective when significant contiguous portions of code are supported.

- **Data types:** Code must use supported data types for acceleration: `double` (both real and complex), `logical`, `char`, `int8`–`32`, `uint8`–`32`. Some `struct`, `cell`, `classdef`, and function handle usage is supported. Sparse arrays are not accelerated.
- **Array shapes:** Array shapes of any size with 3 or fewer dimensions are supported. Changing the shape or data type of an array interrupts acceleration. A few limited situations with 4D arrays are accelerated.
- **Function calls:** Calls to built-in functions and M-functions are accelerated. Calling MEX functions and Java interrupts acceleration. (See also page 14 on inlining simple functions.)
- **Conditionals and loops:** The conditional statements `if`, `elseif`, and simple `switch` statements are supported if the conditional expression evaluates to a scalar. Loops of the form `for k=a:b`, `for k=a:b:c`, and `while` loops are accelerated if all code within the loop is supported.

In-place computation

Introduced in MATLAB 7.3 (R2006b), the element-wise operators (`+`, `.*`, etc.) and some other functions can be computed in-place. That is, a computation like

```
x = 5*sqrt(x.^2 + 1);
```

is handled internally without needing temporary storage for accumulating the result. An M-function can also be computed in-place if its output argument matches one of the input arguments.

```
x = myfun(x);

function x = myfun(x)
x = 5*sqrt(x.^2 + 1);
return;
```

To enable in-place computation, the in-place operation must be within an M-function (and for an in-place function, the function itself must be called within an M-function). Currently, there is no support for in-place computation with MEX-functions.

Multithreaded Computation

MATLAB 7.4 (R2007a) introduced multithreaded computation for multicore and multiprocessor computers. Multithreaded computation accelerates some per-element functions when applied to large arrays (for example, `.^`, `sin`, `exp`) and certain linear algebra functions in the BLAS library. To enable it, select File → Preferences → General → Multithreading and select "Enable multithreaded computation." Further control over parallel computation is possible with the Parallel Computing Toolbox™ using `parfor` and `spmd`.

JIT-Accelerated Example

For example, the following loop-heavy code is supported for acceleration:

```
function B = bilateral(A,sd,sr,R)
% The bilateral image denoising filter

B = zeros(size(A));

for i = 1:size(A,1)
    for j = 1:size(A,2)
        zsum = 0;

        for m = -R:R
            if i+m >= 1 && i+m <= size(A,1)
                for n = -R:R
                    if j+n >= 1 && j+n <= size(A,2)
                        z = exp(-(A(i+m,j+n) - A(i,j))^2/(2*sd^2))...
                            * exp(-(m^2 + n^2)/(2*sr^2));
                        zsum = zsum + z;
                        B(i,j) = B(i,j) + z*A(i+m,j+n);
                    end
                end
            end
        end

        B(i,j) = B(i,j)/zsum;
    end
end
```

For a 128×128 input image and `R` = 3, the run time is 53.3 seconds without acceleration and 0.68 seconds with acceleration.

Vectorization

A computation is *vectorized* by taking advantage of vector operations. A variety of programming situations can be vectorized, and often improving speed to 10 times faster or even better. Vectorization is one of the most general and effective techniques for writing fast M-code.

5.1 Vectorized Computations

Many standard MATLAB functions are "vectorized": they can operate on an array as if the function had been applied individually to every element.

```
>> sqrt([1,4;9,16])

ans =
     1     2
     3     4
```

```
>> abs([0,1,2,-5,-6,-7])

ans =
     0     1     2     5     6     7
```

Consider the following function:

```
function d = minDistance(x,y,z)
% Find the min distance between a set of points and the origin

nPoints = length(x);
d = zeros(nPoints,1);        % Preallocate

for k = 1:nPoints            % Compute distance for every point
   d(k) = sqrt(x(k)^2 + y(k)^2 + z(k)^2);
end

d = min(d);                  % Get the minimum distance
```

For every point, its distance from the origin is computed and stored in `d`. For speed, array `d` is preallocated (see Section 3). The minimum distance is then found with `min`. To vectorize the distance computation, replace the `for` loop with vector operations:

```
function d = minDistance(x,y,z)
% Find the min distance between a set of points and the origin

d = sqrt(x.^2 + y.^2 + z.^2);     % Compute distance for every point
d = min(d);                       % Get the minimum distance
```

The modified code performs the distance computation with vector operations. The `x`, `y` and `z` arrays are first squared using the per-element power operator, `.^` (the per-element operators for multiplication and division are `.*` and `./`). The squared components are added with vector addition. Finally, the square root of the vector sum is computed per element, yielding an array of distances. (A further improvement: it is equivalent to compute `d = sqrt(min(x.^2 + y.^2 + z.^2))`).

The first version of the `minDistance` program takes 0.73 seconds on 50000 points. The vectorized version takes less than 0.04 seconds, more than 18 times faster.

Some useful functions for vectorizing computations:

`min`, `max`, `repmat`, `meshgrid`, `sum`, `cumsum`, `diff`, `prod`, `cumprod`, `accumarray`

5.2 Vectorized Logic

The previous section shows how to vectorize pure computation. But bottleneck code often involves conditional logic. Like computations, MATLAB's logic operators are vectorized:

```
>> [1,5,3] < [2,2,4]

ans =
     1     0     1
```

Two arrays are compared per-element. Logic operations return "logical" arrays with binary values. How is this useful? MATLAB has a few powerful functions for operating on logical arrays:

```
>> find([1,5,3] < [2,2,4]) % Find indices of nonzero elements

ans =
     1     3

>> any([1,5,3] < [2,2,4])  % True if any element of a vector is nonzero
                           % (or per-column for a matrix)
ans =
     1

>> all([1,5,3] < [2,2,4])  % True if all elements of a vector are nonzero
                           % (or per-column for a matrix)
ans =
     0
```

Vectorized logic also works on arrays.

```
>> find(eye(3) == 1)       % Find which elements == 1 in the 3x3 identity matrix

ans =

     1    % (element locations given as indices)
     5
     9
```

By default, `find` returns element locations as indices (see page 16 for indices vs. subscripts).

Example 1: Vector Normalization

To normalize a single vector `v` to unit length, one can use `v = v/norm(v)`. However, to use `norm` for set of vectors `v(:,1)`, `v(:,2)`, ... requires computing `v(:,k)/norm(v(:,k))` in a loop. Alternatively,

```
vMag = sqrt(sum(v.^2));
v = v./vMag(ones(1,size(v,1)),:);
```

Example 2: Removing elements

The situation often arises where array elements must be removed on some per-element condition. For example, this code removes all NaN and infinite elements from an array `x`:

```
i = find(isnan(x) | isinf(x));    % Find bad elements in x
x(i) = [];                        % and delete them
```

Alternatively,

```
i = find(~isnan(x) & ~isinf(x));  % Find elements that are not NaN and not infinite
x = x(i);                         % Keep those elements
```

Both of these solutions can be further streamlined by using logical indexing:

```
x(isnan(x) | isinf(x)) = [];      % Delete bad elements
```

or

```
x = x(~isnan(x) & ~isinf(x));     % Keep good elements
```

Example 3: Piecewise functions

The sinc function has a piecewise definition,

$$\text{sinc}(x) = \begin{cases} \sin(x)/x, & x \neq 0 \\ 1, & x = 0 \end{cases}$$

This code uses `find` with vectorized computation to handle the two cases separately:

```
function y = sinc(x)
% Computes the sinc function per-element for a set of x values.

y = ones(size(x));            % Set y to all ones, sinc(0) = 1
i = find(x ~= 0);             % Find nonzero x values
y(i) = sin(x(i)) ./ x(i);     % Compute sinc where x ~= 0
```

A concise alternative is `y = (sin(x) + (x == 0))./(x + (x == 0))`.

Example 4: Drawing images with meshgrid

The `meshgrid` function takes two input vectors and converts them to matrices by replicating the first over the rows and the second over the columns.

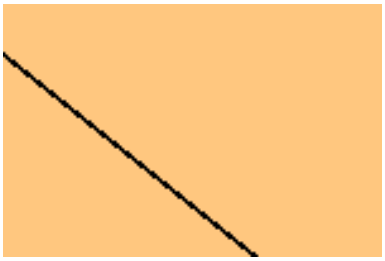
```
>> [x,y] = meshgrid(1:5,1:3)

x =                                                    y =
     1     2     3     4     5                           1     1     1     1     1
     1     2     3     4     5                           2     2     2     2     2
     1     2     3     4     5                           3     3     3     3     3
```

The matrices above work like a map for a width 5, height 3 image. For each pixel, the x-location can be read from `x` and the y-location from `y`. This may seem like a gratuitous use of memory as `x` and `y` simply record the column and row positions, but this is useful. For example, to draw an ellipse,

```
% Create x and y for a width 150, height 100 image
[x,y] = meshgrid(1:150,1:100);

% Ellipse with origin (60,50) of size 15 x 40
Img = sqrt(((x-60).^2 / 15^2) + ((y-50).^2 / 40^2)) > 1;

% Plot the image
imagesc(Img); colormap(copper);
axis image, axis off
```

Drawing lines is just a change in the formula.

```
[x,y] = meshgrid(1:150,1:100);

% The line y = x*0.8 + 20
Img = (abs((x*0.8 + 20) - y) > 1);

imagesc(Img); colormap(copper);
axis image, axis off
```



Polar functions can be drawn by first converting `x` and `y` variables with the `cart2pol` function.

```
[x,y] = meshgrid(1:150,1:100);
[th,r] = cart2pol(x - 75,y - 50);  % Convert to polar

% Spiral centered at (75,50)
Img = sin(r/3 + th);

imagesc(Img); colormap(hot);
axis image, axis off
```

Example 5: Polynomial interpolation

Given n points $x_1, \ldots x_n$ and corresponding function values $y_1, \ldots y_n$, the coefficients $c_0, \ldots, c_{n-1}$ of the interpolating polynomial

$$P(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$$

can be found by solving

$$\begin{bmatrix} {x_1}^{n-1} & {x_1}^{n-2} & \cdots & {x_1}^2 & x_1 & 1 \\ {x_2}^{n-1} & {x_2}^{n-2} & \cdots & {x_2}^2 & x_2 & 1 \\ \vdots & & & & & \vdots \\ {x_n}^{n-1} & {x_n}^{n-2} & \cdots & {x_n}^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} c_{n-1} \\ c_{n-2} \\ \vdots \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

```
function c = polyint(x,y)
% Given a set of points and function values x and y,
% computes the interpolating polynomial.

x = x(:);                               % Make sure x and y are both column vectors
y = y(:);
n = length(x);                          % n = Number of points

% Construct the matrix on the left-hand-side
xMatrix = repmat(x, 1, n);              % Make an n by n matrix with x on every column
powMatrix = repmat(n-1:-1:0, n, 1);     % Make another n by n matrix of exponents
A = xMatrix .^ powMatrix;               % Compute the powers

c = A\y;                                % Solve matrix equation for coefficients
```

The strategy to construct the left-hand side matrix is to first make two `n`×`n` matrices of bases and exponents and then put them together using the per element power operator, `.^` . The `repmat` function ("replicate matrix") is used to make the base matrix `xMatrix` and the exponent matrix `powMatrix`.

$$\texttt{xMatrix} = \begin{bmatrix} \texttt{x(1)} & \texttt{x(1)} & \cdots & \texttt{x(1)} \\ \texttt{x(2)} & \texttt{x(2)} & \cdots & \texttt{x(2)} \\ \vdots & & & \vdots \\ \texttt{x(n)} & \texttt{x(n)} & \cdots & \texttt{x(n)} \end{bmatrix} \qquad \texttt{powMatrix} = \begin{bmatrix} \texttt{n}-1 & \texttt{n}-2 & \cdots & 0 \\ \texttt{n}-1 & \texttt{n}-2 & \cdots & 0 \\ \vdots & & & \vdots \\ \texttt{n}-1 & \texttt{n}-2 & \cdots & 0 \end{bmatrix}$$

The `xMatrix` is made by repeating the column vector `x` over the columns `n` times. The `powMatrix` is a row vector with elements $n-1, n-2, n-3, \ldots, 0$ repeated down the rows `n` times. The two matrices could also have been created with `[powMatrix, xMatrix] = meshgrid(n-1:-1:0, x)`.

The code above is only an example—use the standard `polyfit` function for serious use.

Inlining Simple Functions

Every time an M-file function is called, the MATLAB interpreter incurs some overhead. Additionally, many M-file functions begin with conditional code that checks the input arguments for errors or determines the mode of operation.

Of course, this overhead is negligible for a single function call. It should only be considered when the function being called is an M-file, the function itself is "simple," that is, implemented with only a few lines, and called frequently from within a loop.

For example, this code calls the M-file function `median` repeatedly to perform median filtering:

```
% Apply the median filter of size 5 to signal x
y = zeros(size(x));  % Preallocate

for k = 3:length(x)-2
   y(k) = median(x(k-2:k+2));
end
```

Given a 2500-sample array for `x`, the overall run time is 0.42 seconds.

"Inlining a function" means to replace a call to the function with the function code itself. Beware that inlining should not be confused with MATLAB's "inline" function datatype. Studying `median.m` (type "`edit median`" on the console) reveals that most of the work is done using the built-in `sort` function. The `median` call can be inlined as

```
% Apply the median filter of size 5 to signal x
y = zeros(size(x));  % Preallocate

for k = 3:length(x)-2
   tmp = sort(x(k-2:k+2));
   y(k) = tmp(3);     % y(k) = median(x(k-2:k+2));
end
```

Now the overall run time is 0.047 seconds, nearly 9 times faster. Furthermore, by inlining `median`, it can be specifically tailored to evaluating 5-sample medians. But this is only an example; if the Signal Processing Toolbox™ is available, `y = medfilt1(x,5)` is much faster.

A surprising number of MATLAB's functions are implemented as M-files, of which many can be inlined in just a few lines. In MATLAB 5.3 (R11), the following functions are worth inlining:
`conv`, `cross`, `fft2`, `fliplr`, `flipud`, `ifft`, `ifft2`, `ifftn`, `ind2sub`, `ismember`, `linspace`, `logspace`, `mean`, `median`, `meshgrid`, `poly`, `polyval`, `repmat`, `roots`, `rot90`, `setdiff`, `setxor`, `sortrows`, `std`, `sub2ind`, `union`, `unique`, `var`
The list above is for MATLAB R11; some of these functions have since become built-in. Use the `which` command or try "`edit` *function name*" to determine whether a function is implemented as an M-file.

For example, in MATLAB R11 and earlier, `ifft` was implemented by simply calling `fft` and `conj`. If `x` is a one-dimensional array, `y = ifft(x)` can be inlined with `y = conj(fft(conj(x)))/length(x)`.

Another example: `b = unique(a)` can be inlined with

```
b = sort(a(:));   b( b((1:end-1)') == b((2:end)') ) = [];
```

While `repmat` has the generality to operate on matrices, it is often only necessary to tile a vector or just a scalar. To repeat a column vector `y` over the columns `n` times,

```
A = y(:,ones(1,n));           % Equivalent to A = repmat(y,1,n);
```

To repeat a row vector `x` over the rows `m` times,

```
A = x(ones(1,m),:);           % Equivalent to A = repmat(x,m,1);
```

To repeat a scalar `s` into an `m`×`n` matrix,

```
A = s(ones(m,n));             % Equivalent to A = repmat(s,m,n);
```

This method avoids the overhead of calling an M-file function. It is never slower than `repmat` (critics should note that `repmat.m` itself uses this method to construct `mind` and `nind`). For constructing matrices with constant value, there are other efficient methods, for example, `s+zeros(m,n)`.

Don't go overboard. Inlining functions is only beneficial when the function is simple and when it is called frequently. Doing it unnecessarily obfuscates the code.

7 Referencing Operations

Referencing in MATLAB is varied and powerful enough to deserve a section of discussion. Good understanding of referencing enables vectorizing a broader range of programming situations.

7.1 Subscripts vs. Indices

Subscripts are the most common method used to refer to matrix elements, for example, `A(3,9)` refers to row 3, column 9. Indices are an alternative referencing method. Consider a 10×10 matrix `A`. Internally, MATLAB stores the matrix data linearly as a one-dimensional, 100-element array of data in column-major order.

$$\begin{bmatrix} 1 & 11 & 21 & \cdots & 81 & 91 \\ 2 & 12 & 22 & & 82 & 92 \\ 3 & 13 & 23 & & 83 & 93 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 10 & 20 & 30 & \cdots & 90 & 100 \end{bmatrix}$$

An index refers to an element's position in this one-dimensional array. Using an index, `A(83)` also refers to the element on row 3, column 9.

It is faster to scan down columns than over rows. Column-major order means that elements along a column are sequential in memory while elements along a row are further apart. Scanning down columns promotes cache efficiency.

Conversion between subscripts and indices can be done with the `sub2ind` and `ind2sub` functions. However, because these are M-file functions rather than fast built-in operations, it is more efficient to compute conversions directly (also see Section 6). For a two-dimensional matrix `A` of size `M`×`N`, the conversions between subscript `(i,j)` and index `(index)` are

```
A(i,j)     ↔  A(i + (j-1)*M)
A(index)   ↔  A(rem(index-1,M)+1, floor((index-1)/M)+1)
```

Indexing extends to N-D matrices as well, with indices increasing first through the columns, then through the rows, through the third dimension, and so on. Subscript notation extends intuitively,

`A(..., dim4, dim3, row, col)`.

7.2 Vectorized Subscripts

It is useful to work with submatrices rather than individual elements. This is done with a vector of indices or subscripts. If `A` is a two-dimensional matrix, a vector subscript reference has the syntax

```
A(rowv, colv)
```

where `rowv` and `colv` are vectors. Both may be of any length and their elements may be in any order. If either is a matrix, it is reshaped to a vector. There is no difference between using row vectors or column vectors in vector subscripts.

Let `M = length(rowv)` and `N = length(colv)`. Then a vector subscripted matrix on the right-hand side* returns a submatrix of size `M`×`N`:

$$\begin{bmatrix} \texttt{A(rowv(1),colv(1))} & \texttt{A(rowv(1),colv(2))} & \texttt{A(rowv(1),colv(3))} & \cdots & \texttt{A(rowv(1),colv(N))} \\ \texttt{A(rowv(2),colv(1))} & \texttt{A(rowv(2),colv(2))} & \texttt{A(rowv(2),colv(3))} & \cdots & \texttt{A(rowv(2),colv(N))} \\ \vdots & & & & \vdots \\ \texttt{A(rowv(M),colv(1))} & \texttt{A(rowv(M),colv(2))} & \texttt{A(rowv(M),colv(3))} & \cdots & \texttt{A(rowv(M),colv(N))} \end{bmatrix}$$

A vector subscripted matrix can also appear on the left-hand side as

`A(rowv, colv) =` scalar *or* `M`×`N` matrix

If any elements in the destination reference are repeated, the source element with the greater index dominates. For example,

```
>> A = zeros(2); A([1,2],[2,2]) = [1,2;3,4]

A =
     0     2
     0     4
```

Vector subscript references extend intuitively in higher dimensions.

7.3 Vector Indices

Multiple elements can also be referenced with vector indices.

`A(indexv)`

where `indexv` is an array of indices. On the right-hand side, a vector index reference returns a matrix the same size and shape as `indexv`. If `indexv` is a 3×4 matrix, then `A(indexv)` is the 3×4 matrix

$$\texttt{A(indexv)} = \begin{bmatrix} \texttt{A(indexv(1,1))} & \texttt{A(indexv(1,2))} & \texttt{A(indexv(1,3))} & \texttt{A(indexv(1,4))} \\ \texttt{A(indexv(2,1))} & \texttt{A(indexv(2,2))} & \texttt{A(indexv(2,3))} & \texttt{A(indexv(2,4))} \\ \texttt{A(indexv(3,1))} & \texttt{A(indexv(3,2))} & \texttt{A(indexv(3,3))} & \texttt{A(indexv(3,4))} \end{bmatrix}$$

While vector subscripts are limited to referring to block-shaped submatrices, vector indices can refer to any subset of elements.

If a vector index reference is on the left-hand side, the right-hand side must return a matrix of the same size as `indexv` or a scalar. As with vector subscripts, ambiguous duplicate assignments use the value with greater source index.

*The *left-* and *right-hand sides* (LHS and RHS) refer to the two sides of an assignment, "`LHS = RHS`."

7.4 Reference Wildcards

Using the wildcard, `:`, in a subscript refers to an entire row or column. For example, `A(:,1)` refers to every row in column one–the entire first column. This can be combined with vector subscripts, `A([2,4],:)` refers to the second and fourth rows.

When the wildcard is used in a vector index, the entire matrix is referenced. On the right-hand side, this always returns a column vector.

`A(:)` = column vector

This is frequently useful: for example, if a function input must be a row vector, the user's input can be quickly reshaped into row vector with `A(:).'` (make a column vector and transpose to a row vector).

`A(:)` on the left-hand side assigns all the elements of `A`, but does not change its size. For example, `A(:) = 8` changes all elements of matrix `A` to 8.

7.5 Logical Indexing

Given a logical array `mask` with the same size as `A`,

```
A(mask)
```

refers to all elements of `A` where the corresponding element of `mask` is 1 (true). It is equivalent to `A(find(mask))`. A common usage of logical indexing is to refer to elements based on a per-element condition, for example,

```
A(abs(A) < 1e-3) = 0
```

sets all elements with magnitude less than 10^{-3} to zero. Logical indexing is also useful to select non-rectangular sets of elements, for example,

```
A(logical(eye(size(A))))
```

references the diagonal elements of `A`. Note that for a right-hand side reference, it is faster to use `diag(A)` to get the diagonal of `A`.

7.6 Deleting Submatrices with []

Elements in a matrix can be deleted by assigning the empty matrix. For example, `A([3,5]) = []` deletes the third and fifth element from `A`. If this is done with index references, the matrix is reshaped into a row vector.

It is also possible to delete with subscripts if all but one subscript are the wildcard. `A(2,:) = []` deletes the second row. Deletions like `A(2,1) = []` or even `A(2,1:end) = []` are illegal.

8 Solving $Ax = b$

Many problems involve solving a linear system, that is, solving the $Ax = b$ problem

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

The *condition number* $\text{cond}(A)$ describes how sensitive the problem is to small perturbations. MATLAB can efficiently approximate $\text{cond}(A)$ with `condest`. Smaller condition number is better, and in this case A is said to be well-conditioned. An infinite condition number implies that A is *singular*—the problem cannot be solved, though it may be solved approximately with the pseudoinverse (`x = pinv(A)*b`).

Small- to moderately-sized problems are solved efficiently by the backslash operator:

```
x = A\b;    % Solves A*x = b
```

We shall focus on large problems.

8.1 Iterative Methods

For large problems, MATLAB is well-equipped with iterative solvers: `bicg`, `bicgstab`, `cgs`, `gmres`, `lsqr`, `minres`, `pcg`, `symmlq`, `qmr`. The general calling syntax (except for `gmres`) is

`[x,flag,relres,iter,resvec] =` *method*`(A,b,tol,maxit,M1,M2,x0)`

where the required arguments are the matrix `A`, the right-hand side `b`, and the solution `x`. The `minres`, `pcg`, and `symmlq` methods require `A` to be symmetric (`A = A'`) and `pcg` additionally requires positive definiteness (all eigenvalues are positive). The optional input variables are

`tol`	accuracy tolerance (default 10^{-6})
`maxit`	maximum number of iterations (default 20)
`M1`, `M2`	the method uses preconditioner `M = M1*M2`; it solves $M^{-1}Ax = M^{-1}b$
`x0`	initial guess

The optional output variables are

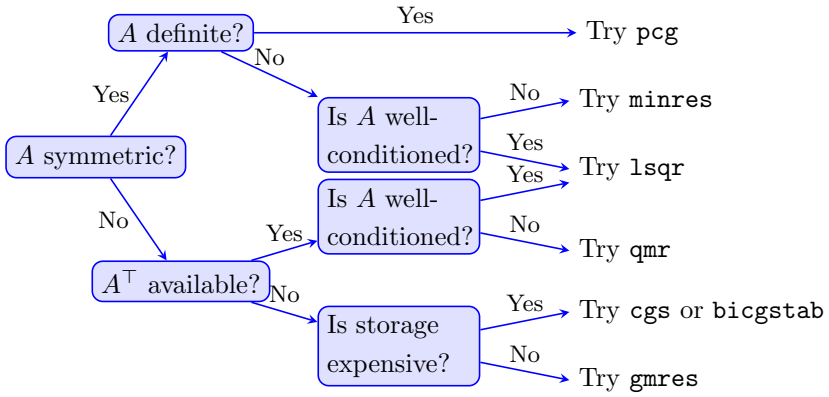
`flag`	convergence flag (0 = success, nonzero values are error codes)
`relres`	relative residual $\|b - Ax\|/\|b\|$
`iter`	number of iterations
`resvec`	a vector listing the residual norm $\|b - Ax\|$ at each iteration

The `gmres` method includes one additional input,

`[x,flag,relres,iter,resvec] = gmres(A,b,restart,tol,maxit,...)`

The method restarts every `restart` iterations, or if `restart=[]`, the method does not restart.

Which method works best depends on the particular problem. This diagram (adapted from Demmel [3]) provides a reasonable starting point.



Functional Representation

Rather than storing the matrix explicitly, it is often convenient to represent A as a linear function acting on x. For example,

$$\texttt{A*x} = \begin{bmatrix} 1 & & & \\ 1 & 1 & & \\ \vdots & & \ddots & \\ 1 & 1 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

can be expressed as `A*x = cumsum(x)`. In addition to being conceptually convenient, functional representation avoids the memory overhead of storing the matrix.

All the iterative solvers support functional representation, where a function handle `afun` is given in place of matrix `A`. The function `afun(x)` should accept a vector `x` and return `A*x`. The help for `lsqr` ("`help lsqr`") has a functional example where `A` is the central differencing operator. The `bicg`, `lsqr`, and `qmr` methods also require the transpose: `afun(x,'notransp')` should return `A*x` and `afun(x,'transp')` should return `A'*x`. Similarly, the preconditioner may be given as a function handle `mfun`.

Special Matrices

For certain special matrix structures, the $Ax = b$ problem can be solved very quickly.

- *Circulant matrices.* Matrices corresponding to cyclic convolution $Ax = h * x$ are diagonalized in the Fourier domain, and can be solved by `x = ifft(fft(b)./fft(h));`
- *Triangular and banded.* Triangular matrices and diagonally-dominant banded matrices are solved efficiently by sparse LU factorization: `[L,U] = lu(sparse(A)); x = U\(L\b);`
- *Poisson problem.* If A is a finite difference approximation of the Laplacian, the problem is efficiently solved by multigrid methods (e.g., web search for "matlab multigrid").

8.2 Inlined PCG

As discussed in Section 6, it is sometimes advantageous to inline a simple piece of code to tailor it to a specific application. While most iterative methods are too lengthy for inlining, conjugate gradients has a short implementation. The value of inlining `pcg` is that calls to `afun` and `mfun` may be inlined. Beware that in the following code, A must be symmetric positive definite—otherwise, it will not work!

```
%% Use PCG to solve Ax = b %%
tol = 1e-6;   % Convergence tolerance
maxit = 20;   % Maximum number of iterations
x = x0;       % Set x to initial guess  (if no guess is available, use x0 = 0)

Compute w = A x

r = b - w;            % r = residual vector

Set p = r, or if using a preconditioner, p = M^-1 r

d = r'*p;

for iter = 1:maxit
    if norm(p) < tol
        break;        % PCG converged successfully (small change in x)
    end

    Compute w = A p

    alpha = e/(p'*w);
    x = x + alpha*p;
    r = r - alpha*w;

    Set w = r, or if using a preconditioner, w = M^-1 r

    if norm(w) < tol && norm(r) < tol
        break;        % PCG converged successfully (small residual)
    end

    dlast = d;
    d = r'*w;
    p = w + (d/dlast)*p;
end
```

After the loop, the final solution is in `x`. The method converged successfully with accuracy `tol` if it encountered a `break` statement, otherwise, it failed or needed more iterations. The relative residual may be computed by `relres = norm(r)/norm(b)` after the loop and `resvec` by storing the value of `norm(r)` each iteration.

9 Numerical Integration

Quadrature formulas are numerical approximation of integrals of the form

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_k w_k f(x_k),$$

where the x_k are called the *nodes* or *abscissas* and w_k are the associated *weights*. Simpson's rule is

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\left[f(a) + 4f(a+h) + f(b)\right], \qquad h = \frac{b-a}{2}.$$

Simpson's rule is a quadrature formula with nodes a, $a+h$, b and node weights $\frac{h}{3}$, $\frac{4h}{3}$, $\frac{h}{3}$.

MATLAB has several functions for quadrature in one dimension:

`quad`	adaptive Simpson	Better for low accuracy and nonsmooth integrands
`quadl`	adaptive Gauss-Lobatto	Higher accuracy with smoother integrands
`quadgk`	adaptive Gauss-Kronrod	Oscillatory integrands and high accuracy
`quadv`	adaptive Simpson	Vectorized for multi-valued integrands

The `quad-` functions are robust and precise, however, they are not very efficient. They use an adaptive refinement procedure, and while this reduces the number of function calls, they gain little from vectorization and incur significant overhead.

If an application requires approximating an integral whose integrand can be efficiently vectorized, using nonadaptive quadrature may improve speed.

```
for n = -20:20  % Compute Fourier series coefficients
    c(n + 21) = quad(@(x)(exp(sin(x).^6).*exp(-1i*x*n)),0,pi,1e-4);
end
```

This code runs in 5.16 seconds. In place of `quad`, using Simpson's composite rule with `N = 199` nodes yields results with comparable accuracy and allows for vectorized computation. Since the integrals are all over the same interval, the nodes and weights only need to be constructed once.

```
N = 199;  h = pi/(N-1);
x = (0:h:pi).';                                          % Nodes
w = ones(1,N);  w(2:2:N-1) = 4;  w(3:2:N-2) = 2;  w = w*h/3;    % Weights

for n = -20:20
    c(n + 21) = w * ( exp(sin(x).^6).*exp(-1i*x*n) );
end
```

This version of the code runs in 0.02 seconds (200 times faster). The quadrature is performed by the dot product multiplication with `w`. It can be further optimized by replacing the `for` loop with one vector-matrix multiply:

```
[n,x] = meshgrid(-20:20, 0:h:pi);
c = w * ( exp(sin(x).^6).*exp(-1i*x.*n) );
```

For this example, `quadv` can be used on a multi-valued integrand with similar accuracy and speed,

```
n = −20:20;
c = quadv(@(x)exp(sin(x).^6).*exp(−1i.*x.*n),0,pi,1e−4);
```

9.1 One-Dimensional Integration

$\int_a^b f(x)\,\mathrm{d}x$ is approximated by composite Simpson's rule with

```
h = (b − a)/(N−1);
x = (a:h:b).';
w = ones(1,N);  w(2:2:N−1) = 4;  w(3:2:N−2) = 2;  w = w*h/3;
I = w * f(x);    % Approximately evaluate the integral
```

where `N` is an odd integer specifying the number of nodes.

A good higher-order choice is 4th-order Gauss-Lobatto [4] (as used by `quadl`), based on

$$\int_{-1}^{1} f(x)\,\mathrm{d}x \approx \tfrac{1}{6}f(-1) + \tfrac{5}{6}f(-\tfrac{1}{\sqrt{5}}) + \tfrac{5}{6}f(\tfrac{1}{\sqrt{5}}) + \tfrac{1}{6}f(1).$$

```
N = max(3*round((N−1)/3),3) + 1;  % Adjust N to the closest valid choice
h = (b − a)/(N−1);
d = (3/sqrt(5) − 1)*h/2;
x = (a:h:b).';  x(2:3:N−2) = x(2:3:N−2) − d;  x(3:3:N−1) = x(3:3:N−1) + d;
w = ones(1,N);  w(4:3:N−3) = 2;  w([2:3:N−2,3:3:N−1]) = 5;  w = w*h/4;
I = w * f(x);    % Approximately evaluate the integral
```

The number of nodes `N` must be such that `(N − 1)/3` is an integer. If not, the first line adjusts `N` to the closest valid choice. It is usually more accurate than Simpson's rule when f has six continuous derivatives, $f \in C^6(a,b)$.

A disadvantage of this nonadaptive approach is that the accuracy of the result is only indirectly controlled by the parameter `N`. To guarantee a desired accuracy, either use a generously large value for `N` or, if possible, determine the error bounds [6, 7]

$$\begin{aligned} \textit{Simpson's rule error} &\le \frac{(b-a)h^4}{180} \max_{a\le\eta\le b} \left|f^{(4)}(\eta)\right| \quad \text{provided } f \in C^4(a,b), \\ \textit{4th-order Gauss-Lobatto error} &\le \frac{27(b-a)h^6}{56000} \max_{a\le\eta\le b} \left|f^{(6)}(\eta)\right| \quad \text{provided } f \in C^6(a,b), \end{aligned}$$

where $h = \frac{b-a}{N-1}$. Note that these bounds are valid only when the integrand is sufficiently differentiable: f must have four continuous derivatives for the Simpson's rule error bound, and six continuous derivatives for 4th-order Gauss-Lobatto.

Composite Simpson's rule is a fast and effective default choice. But depending on the situation, other methods may be faster and more accurate:

- If the integrand is expensive to evaluate, or if high accuracy is required and the error bounds above are too difficult to compute, use one of MATLAB's adaptive methods. Otherwise, consider composite methods.
- Use higher-order methods (like Gauss-Lobatto/`quadl`) for very smooth integrands and lower-order methods for less smooth integrands.
- Use the substitution $u = \frac{1}{1-x}$ or Gauss-Laguerre quadrature for infinite-domain integrals like $\int_0^\infty$.

9.2 Multidimensional Integration

An approach for evaluating double integrals of the form $\int_a^b \int_c^d f(x,y)\,\mathrm{d}y\,\mathrm{d}x$ is to apply one-dimensional quadrature to the outer integral $\int_a^b F(x)\,\mathrm{d}x$ and then for each x use one-dimensional quadrature over the inner dimension to approximate $F(x) = \int_c^d f(x,y)\,\mathrm{d}y$. The following code does this with composite Simpson's rule with `Nx`×`Ny` nodes:

```
%%% Construct Simpson nodes and weights over x %%%
h = (b - a)/(Nx-1);
x = (a:h:b).';
wx = ones(1,Nx);  wx(2:2:Nx-1) = 4;  wx(3:2:Nx-2) = 2;  wx = w*h/3;

%%% Construct Simpson nodes and weights over y %%%
h = (d - c)/(Ny-1);
y = (c:h:d).';
wy = ones(1,Ny);  wy(2:2:Ny-1) = 4;  wy(3:2:Ny-2) = 2;  wy = w*h/3;

%%% Combine for two-dimensional integration %%%
[x,y] = meshgrid(x,y);  x = x(:);  y = y(:);
w = wy.'*wx;  w = w(:).';

I = w * f(x,y);    % Approximately evaluate the integral
```

Similarly for three-dimensional integrals, the weights are combined with

```
[x,y,z] = meshgrid(x,y,z);  x = x(:);  y = y(:);  z = z(:);
w = wy.'*wx;  w = w(:)*wz;  w = w(:).';
```

When the integration region is complicated or of high dimension, Monte Carlo integration techniques are appropriate. The disadvantage is that an N-point Monte Carlo quadrature has error on the order $O(\frac{1}{\sqrt{N}})$, so many points are necessary even for moderate accuracy. Suppose that N points, $x_1, x_2, \ldots, x_N$, are uniformly randomly selected in a multidimensional region (or volume) Ω. Then

$$\int_\Omega f\,\mathrm{d}V \approx \frac{\int_\Omega \mathrm{d}V}{N} \sum_{n=1}^{N} f(x_n).$$

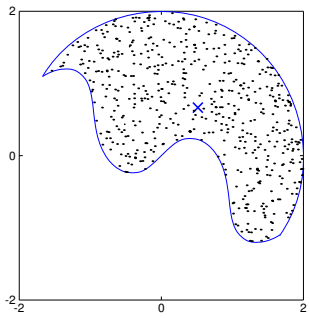
To integrate a complicated region W that is difficult to sample uniformly, find an easier region Ω that contains W and can be sampled [5]. Then

$$\int_W f \, dV = \int_\Omega f \cdot \chi_W \, dV \approx \frac{\int_\Omega dV}{N} \sum_{n=1}^{N} f(x_n)\chi_W(x_n), \qquad \chi_W(x) = \begin{cases} 1, & x \in W, \\ 0, & x \notin W. \end{cases}$$

$\chi_W(x)$ is the *indicator function* of W: $\chi_W(x) = 1$ when x is within region W and $\chi_W(x) = 0$ otherwise. Multiplying the integrand by χ_W sets contributions from outside of W to zero.

For example, consider finding the center of mass of the shape W defined by $\cos\left(2\sqrt{x^2+y^2}\right) x \le y$ and $x^2 + y^2 \le 4$. Given the integrals $M = \int_W dA$, $M_x = \int_W x \, dA$, and $M_y = \int_W y \, dA$, the center of mass is $(\frac{M_x}{M}, \frac{M_y}{M})$. The region is contained in the rectangle Ω defined by $-2 \le x \le 2$ and $-2 \le y \le 2$. The following code estimates M, M_x, and M_y:

```
%%% Uniformly randomly sample points (x,y) in Ω %%%
x = 4*rand(N,1) - 2;
y = 4*rand(N,1) - 2;

%%% Restrict the points to region W %%%
i = (cos(2*sqrt(x.^2 + y.^2)).*x <= y & x.^2 + y.^2 <= 4);
x = x(i);  y = y(i);

%%% Approximately evaluate the integrals %%%
area = 4*4;          % The area of rectangle Ω
M = (area/N) * length(x);
Mx = (area/N) * sum(x);
My = (area/N) * sum(y);
```



Region W sampled with $N = 1500$. The center of mass is $\approx (0.5, 0.7)$.

More generally, if W is a two-dimensional region contained in the rectangle defined by $a \le x \le b$ and $c \le y \le d$, the following code approximates $\int_W f \, dA$:

```
x = a + (b-a)*rand(N,1);
y = c + (d-c)*rand(N,1);
i = logical(indicatorW(x,y));
x = x(i);  y = y(i);

area = (b-a)*(d-c);
I = (area/N) * sum(f(x,y));    % Approximately evaluate the integral
```

where `indicatorW(x,y)` is the indicator function $\chi_W(x, y)$ for region W.

For refinements and variations of Monte Carlo integration, see for example [1].

10 Signal Processing

Even without the Signal Processing Toolbox, MATLAB is quite capable in signal processing computations. This section lists code snippets to perform several common operations efficiently.

Moving-average filter

To compute an `N`-sample moving average of `x` with zero padding:

```
y = filter(ones(N,1)/N,1,x);
```

For large `N`, it is faster to use

```
y = cumsum(x)/N;
y(N+1:end) = y(N+1:end) - y(1:end-N);
```

Locating zero-crossings and extrema

To obtain the indices where signal `x` crosses zero:

```
i = find(diff(sign(x)));
% The kth zero-crossing lies between x(i(k)) and x(i(k)+1)
```

Linear interpolation can be used for subsample estimates of zero-crossing locations:

```
i = find(diff(sign(x)));
i = i - x(i)./(x(i+1) - x(i));  % Linear interpolation
```

Since local maximum and minimum points of a signal have zero derivative, their locations can be estimated from the zero-crossings of `diff(x)`, provided the signal is sampled with sufficiently fine resolution. For a coarsely sampled signal, a better estimate is

```
iMax = find(sign(x(2:end-1)-x(1:end-2)) ...
          + sign(x(2:end-1)-x(3:end)) > 0) + 1;
iMin = find(sign(x(2:end-1)-x(1:end-2)) ...
          + sign(x(2:end-1)-x(3:end)) < 0) + 1;
```

FFT-based convolution

This line performs FFT-based circular convolution, equivalent to `y = filter(b,1,x)` except near the boundaries, provided that `length(b)` < `length(x)`:

```
y = ifft(fft(b,length(x)).*fft(x));
```

For FFT-based zero-padded convolution, equivalent to `y = filter(b,1,x)`,

```
N = length(x)+length(b)-1;
y = ifft(fft(b,N).*fft(x,N));
y = y(1:length(x));
```

In both code snippets above, `y` will be complex-valued, even if `x` and `b` are both real. To force `y` to be real, follow the computation with `y = real(y)`. If you have the Signal Processing Toolbox, it is faster to use `fftfilt` for FFT-based, zero-padded filtering.

Noncausal filtering and other boundary extensions

For its intended use, the `filter` command is limited to causal filters, that is, filters that do not involve "future" values to the right of the current sample. Furthermore, `filter` is limited to zero-padded boundary extension; data beyond the array boundaries are assumed zero as if the data were padded with zeros.

For two-tap filters, noncausal filtering and other boundary extensions are possible through `filter`'s fourth initial condition argument. Given a boundary extension value `padLeft` for `x(0)`, the filter $y_n = b_1 x_n + b_2 x_{n-1}$ may be implemented as

```
y = filter(b,1,x,padLeft*b(2));
```

Similarly, given a boundary extension value `padRight` for `x(end+1)`, the filter $y_n = b_1 x_{n+1} + b_2 x_n$ is implemented as

```
y = filter(b,1,[x(2:end),padRight],x(1)*b(2));
```

Choices for `padLeft` and `padRight` for various boundary extensions are

Boundary extension	`padLeft`	`padRight`
Periodic	`x(end)`	`x(1)`
Whole-sample symmetric	`x(2)`	`x(end-1)`
Half-sample symmetric	`x(1)`	`x(end)`
Antisymmetric	`2*x(1)-x(2)`	`2*x(end)-x(end-1)`

It is in principle possible to use a similar approach for longer filters, but ironically, computing the initial condition itself requires filtering. To implement noncausal filtering and filtering with other boundary handling, it is usually fastest to pad the input signal, apply `filter`, and then truncate the result.

Upsampling and Downsampling

Upsample `x` (insert zeros) by factor `p`:

```
y = zeros(length(x)*p-p+1,1);  % For trailing zeros, use y = zeros(length(x)*p,1);
y(1:p:length(x)*p) = x;
```

Downsample `x` by factor `p`, where $1 \leq$ `q` $\leq$ `p`:

```
y = x(q:p:end);
```

Haar Wavelet

This code performs `K` stages of the Haar wavelet transform on the input data `x` to produce wavelet coefficients `y`. The input array `x` must have length divisible by 2^K.

Forward transform:

```
y = x(:); N = length(y);

for k = 1:K
   tmp = y(1:2:N) + y(2:2:N);
   y([1:N/2,N/2+1:N]) = ...
      [tmp;y(1:2:N) - 0.5*tmp]/sqrt(2);
   N = N/2;
end
```

Inverse transform:

```
x = y(:); N = length(x)*pow2(-K);

for k = 1:K
   N = N*2;
   tmp = x(N/2+1:N) + 0.5*x(1:N/2);
   x([1:2:N,2:2:N]) = ...
      [tmp;x(1:N/2) - tmp]*sqrt(2);
end
```

Daubechies 4-Tap Wavelet

This code implements the Daubechies 4-tap wavelet in lifting scheme form [2]. The input array `x` must have length divisible by 2^K. Filtering is done with symmetric boundary handling.

Forward transform:

```
U = 0.25*[2-sqrt(3),-sqrt(3)];
ScaleS = (sqrt(3) - 1)/sqrt(2);
ScaleD = (sqrt(3) + 1)/sqrt(2);

y = x(:); N = length(y);

for k = 1:K
   N = N/2;
   y1 = y(1:2:2*N);
   y2 = y(2:2:2*N) + sqrt(3)*y1;
   y1 = y1 + filter(U,1,...
      y2([2:N,max(N-1,1)]),y2(1)*U(2));
   y(1:2*N) = [ScaleS*...
      (y2 - y1([min(2,N),1:N-1]));ScaleD*y1];
end
```

Inverse transform:

```
U = 0.25*[2-sqrt(3),-sqrt(3)];
ScaleS = (sqrt(3) - 1)/sqrt(2);
ScaleD = (sqrt(3) + 1)/sqrt(2);

x = y(:); N = length(x)*pow2(-K);

for k = 1:K
   y1 = x(N+1:2*N)/ScaleD;
   y2 = x(1:N)/ScaleS + y1([min(2,N),1:N-1]);
   y1 = y1 - filter(U,1,...
      y2([2:N,max(N-1,1)]),y2(1)*U(2));
   x([1:2:2*N,2:2:2*N]) = [y1;y2 - sqrt(3)*y1];
   N = 2*N;
end
```

To use periodic boundary handling rather than symmetric boundary handling, change appearances of `[2:N,max(N-1,1)]` to `[2:N,1]` and `[min(2,N),1:N-1]` to `[N,1:N-1]`.

Miscellaneous

Clip a value without using if statements

To clip (or clamp, bound, or saturate) a value to within a range, the straightforward implementation is

```
if x < lowerBound
   x = lowerBound;
elseif x > upperBound
   x = upperBound;
end
```

Unfortunately, this is slow. A faster method is to use the `min` and `max` functions:

```
x = max(x,lowerBound);      % Clip elements from below, force x >= lowerBound
x = min(x,upperBound);      % Clip elements from above, force x <= upperBound
```

Moreover, it works per-element if `x` a matrix of any size.

Convert any array into a column vector

It is often useful to force an array to be a column vector, for example, when writing a function expecting a column vector as an input. This simple trick will convert the input array of any shape and number of dimensions to a column vector, even if the input is already a column vector.

```
x = x(:);      % Convert x to a column vector
```

Similarly, `x = x(:,:)` reduces an N-d array to 2D, where the number of rows stays the same.

Find the min/max of a matrix or N-d array

Given a matrix input (with no other inputs), the `min` and `max` functions operate along the columns, finding the extreme element in each column. To find the extreme element over the entire matrix, one way for a two-dimensional matrix is `min(min(A))`. A better method that works regardless of the number of dimensions and determines the extreme element's location is

```
[MinValue, MinIndex] = min(A(:));    % Find the minimum element in A
                                     % The minimum value is MinValue, the index is MinIndex
MinSub = ind2sub(size(A), MinIndex); % Convert MinIndex to subscripts
```

The minimum element is `A(MinIndex)` or `A(MinSub(1), MinSub(2), ...)` as a subscript reference. (Similarly, replace `min` with `max` for maximum value.)

Flood filling

Flood filling, like the "bucket" tool in image editors, can be elegantly written as a recursive function:

```
function I = flood1(I,c,x,y)
% Flood fills image I from point (x,y) with color c.

c2 = I(y,x);
I(y,x) = c;

if x > 1         & I(y,x-1) == c2 & I(y,x-1) ~= c, I = flood1(I,c,x-1,y); end
if x < size(I,2) & I(y,x+1) == c2 & I(y,x+1) ~= c, I = flood1(I,c,x+1,y); end
if y > 1         & I(y-1,x) == c2 & I(y-1,x) ~= c, I = flood1(I,c,x,y-1); end
if y < size(I,1) & I(y+1,x) == c2 & I(y+1,x) ~= c, I = flood1(I,c,x,y+1); end
```

Being a highly recursive function, this is inefficient in MATLAB. The following code is faster:

```
function I = flood2(I,c,x,y)
% Flood fills image I from point (x,y) with color c.

LastFlood = zeros(size(I));
Flood = LastFlood;
Flood(y,x) = 1;
Mask = (I == I(y,x));
FloodFilter = [0,1,0; 1,1,1; 0,1,0];

while any(LastFlood(:) ~= Flood(:))
   LastFlood = Flood;
   Flood = conv2(Flood,FloodFilter,'same') & Mask;
end

I(find(Flood)) = c;
```

The key is the `conv2` two-dimensional convolution function. Flood filling a 40×40-pixel region takes 1.168 seconds with `flood1` and 0.067 seconds with `flood2`.

Vectorized use of set on GUI objects

While not a speed improvement, a trick for cleaner code is vectorized use of `set` on GUI objects. A serious graphical user interface (GUI) can have dozens of objects and many properties to configure. For example, to define three edit boxes with white text background and left text alignment:

```
uicontrol('Units', 'normalized', 'Position', [0.1,0.9,0.7,0.05], ...
   'HorizontalAlignment', 'left', 'Style', 'edit', 'BackgroundColor', [1,1,1]);
uicontrol('Units', 'normalized', 'Position', [0.1,0.8,0.7,0.05], ...
   'HorizontalAlignment', 'left', 'Style', 'edit', 'BackgroundColor', [1,1,1]);
uicontrol('Units', 'normalized', 'Position', [0.1,0.7,0.7,0.05], ...
   'HorizontalAlignment', 'left', 'Style', 'edit', 'BackgroundColor', [1,1,1]);
```

A vectorized call to `set` reduces this to

```
h(1) = uicontrol('Units', 'normalized', 'Position', [0.1,0.9,0.7,0.05]);
h(2) = uicontrol('Units', 'normalized', 'Position', [0.1,0.8,0.7,0.05]);
h(3) = uicontrol('Units', 'normalized', 'Position', [0.1,0.7,0.7,0.05]);

set(h, 'HorizontalAlignment', 'left', 'Style', 'edit','BackgroundColor', [1,1,1]);
```

12 Further Reading

For more reading on vectorization, see the MathWorks vectorization guide:

`http://www.mathworks.com/support/tech-notes/1100/1109.shtml`

For further details and examples on JIT Acceleration, see for example

`http://www.mathworks.com/access/helpdesk_r13/help/techdoc/matlab_prog/ch7_perf.html`

For a thorough guide on efficient array manipulation, see *MATLAB array manipulation tips and tricks*:

`http://home.online.no/~pjacklam/matlab/doc/mtt`

For numerical methods with MATLAB, see *Numerical Computing with MATLAB*:

`http://www.mathworks.com/moler`

MEX

In a coding situation that cannot be optimized any further, keep in mind that the MATLAB language is intended primarily for easy prototyping rather than high-speed computation. In some cases, an appropriate solution is Matlab Executable (MEX) external interface functions. With a C or Fortran compiler, it is possible to produce a MEX function that can be called from within MATLAB in the same as an M-function. The typical speed improvement over equivalent M-code is easily ten-fold.

Installations of MATLAB include an example of MEX under directory `<MATLAB>/extern/examples/mex`. In this directory, a function "`yprime`" is written as M-code (`yprime.m`) and equivalent C (`yprime.c`) and Fortran (`yprime.f`) MEX source code. For more information, see the MathWorks MEX-files guide

`http://www.mathworks.com/support/tech-notes/1600/1605.html`

For information on compiling MEX files with the GNU compiler collection (GCC), see

`http://gnumex.sourceforge.net`

Beware, MEX is an ambitious alternative to M-code. Writing MEX files requires learning some low-level details about MATLAB, and takes more time to develop than M-code.

Matlab Compiler

The MATLAB Compiler generates a standalone executable from M-code. For product information and documentation, see

`http://www.mathworks.com/access/helpdesk/help/toolbox/compiler`

References

[1] A. BIELAJEW. "The Fundamentals of the Monte Carlo Method for Neutral and Charged Particle Transport." *University of Michigan, class notes.* Available online at
`http://www-personal.engin.umich.edu/~bielajew/MCBook/book.pdf`

[2] I. DAUBECHIES AND W. SWELDENS. "Factoring Wavelet Transforms into Lifting Steps." Sep. 1996.

[3] J. W. DEMMEL. *Applied Numerical Linear Algebra.* SIAM, 1997.

[4] W. GANDER AND W. GAUTSCHI. "Adaptive Quadrature–Revisited," *BIT*, vol. 40, pp. 84-101, 2000.

[5] W. PRESS, B. FLANNERY, S. TEUKOLSKY AND W. VETTERLING. *Numerical Recipes.* Cambridge University Press, 1986.

[6] E. WEISSTEIN. "Lobatto Quadrature." From MathWorld–A Wolfram Web Resource.
`http://mathworld.wolfram.com/LobattoQuadrature.html`

[7] E. WEISSTEIN. "Simpson's Rule." From MathWorld–A Wolfram Web Resource.
`http://mathworld.wolfram.com/SimpsonsRule.html`