Writing Fast MATLAB Code

Pascal Getreuer, June 2006

Contents

Introduction

The MATLAB programming language is parsed; code is interpreted during runtime. Languages like C and Fortran are faster because they are first compiled into the computer's native language. The advantages of parsing in realtime are greater platform independence, greater language flexibility, and easier debugging. The disadvantages are slower speed, increased overhead, and limited low-level control.

This article discusses strategies for improving the speed of MATLAB code. Keep in mind that MATLAB has gone through many versions and that it is available on many platforms. The fastest method on one system may not be the fastest on another. This article provides methods that are generally fast, but makes no claim on what is *fastest*.

Caution!

- **Learn the language first.** Optimization requires comfort with the syntax and functions of the language. **This article is not a tutorial on MATLAB.**
- **Use comments.** Optimized code tends to be terse and cryptic. Help others and yourself by remembering to comment.
- **Don't optimize code before its time.** Is optimization worth the effort? If the code will soon be revised or extended, it will be rewritten anyway.
- **Only optimize where necessary.** Focus your efforts on bottlenecks.

1 The Profiler

MATLAB 5.0 and newer versions include a tool called the "profiler" that helps determine where the bottlenecks are in a program. Consider 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 the profiler and clear any old profiler data:

```
>> profile on
>> profile clear
```

Now run the program. Change the input argument higher or lower so that it takes about a second.

```
>> example1(5000);
```

Then enter

```
>> profile report
```

The profiler generates an HTML report on the function and launches a browser window. Depending on the system, profiler results may be a little different 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:

100% of the total time in this function was spent on the following lines:

```
                3:  for k = 1:Count
 2.11   68%     4:      result(k) = sin(k/50);
                5:
 0.14    5%     6:      if result(k) < -0.9
 0.84   27%     7:          result(k) = gammaln(k);
                8:      end
```

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.

2 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 `for` 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
```

The profiler timed this code to take 0.47 seconds. 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? Then use the upper bound on the array size and cut the excess after the loop:

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

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

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

a = a(1:count);         % Trim the result
```

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

Preallocation can also be done with cell arrays, using the `cell` command to create the desired size. Using preallocation with a frequently resizing cell array is even more beneficial than with double arrays.

3 Vectorization

To "vectorize" a computation means to replace parallel operations with vector operations. This strategy often improves speed ten-fold. Good vectorization is a skill that must be developed; it requires comfort with MATLAB's language and creativity.

3.1 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, the distance between it and the origin is computed and stored in `d`. 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.

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`, `filter`

3.2 Vectorized Logic

The previous section shows how to vectorize pure computation. 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 elements.

How is this useful? MATLAB has a few powerful functions for operating on logical arrays:

- `find`: Find indices of nonzero elements.
- `any`: True if any element of a vector is nonzero (or per-column for a matrix).
- `all`: True if all elements of a vector are nonzero (or per-column for a matrix).

```
>> find([1,5,3] < [2,2,4])

ans =
     1     3
```

```
>> find(eye(3) == 1)

ans =
     1
     5
     9
```

The `find` function returns the indices where the vector logic operation returns true. In the first example, $1 < 2$ is true, $5 < 2$ is false, and $3 < 4$ is true, so `find` reports that the first and third comparisons are true. In the second example, `find` returns the indices where the identity matrix is equal to one. The indices 1, 5, and 9 correspond to the diagonal of a 3 by 3 matrix.

The `any` and `all` functions are simple but occasionally very useful. For example, `any(x(:) < 0)` returns true if any element of `x` is negative.

Example 1: 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 2: Piecewise functions

The sinc function has a piecewise definition,

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

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
```

An interesting alternative is `y = (sin(x) + (x == 0))./(x + (x == 0))`.

Example 3: 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 =
     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5
```

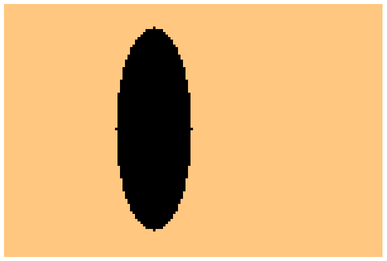
```
y =
     1     1     1     1     1
     2     2     2     2     2
     3     3     3     3     3
```

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;
```

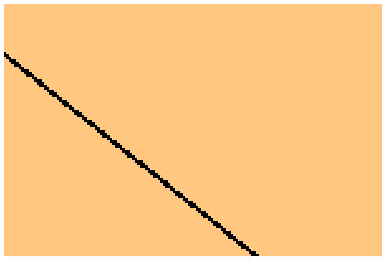


Drawing lines is almost the same, just a change in the formula.

```
[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;
```

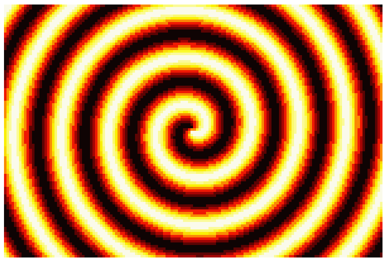


Polar functions can be drawn by first converting `x` and `y` variables with the `cart2pol` function.

```
[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 4: Polynomial interpolation

Given n points $x_1, x_2, x_3, \ldots x_n$ and n corresponding function values $y_1, y_2, y_3, \ldots y_n$, the coefficients $c_0, c_1, c_2, \ldots c_{n-1}$ of the interpolating polynomial of degree $n-1$ 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 left-hand side matrix %%%
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` by `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)`.

This function is only an example; use the standard `polyfit` function for serious polynomial interpolation. It is more general and algorithmically more efficient (see `polyfit.m`).

4 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 for a 2500-sample input 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. If called repeatedly, the following functions are worthwhile inlining:

- `linspace`, `logspace`
- `mean`, `median`, `std`, `var`

- `ifft`, `fft2`, `ifft2`, `ifftn`, `conv`
- `fliplr`, `flipud`, `meshgrid`, `repmat`, `rot90`, `sortrows`
- `ismember`, `setdiff`, `setxor`, `union`, `unique`
- `poly`, `polyval`, `roots`
- `sub2ind`, `ind2sub`

(To view the code for a function, type "`edit` *function name*" on the console).

For example, `ifft` is implemented by simply calling `fft` and `conj` (see `ifft.m`). 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(find(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` by `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)`.

Warning: 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.

5 Numerical Integration

Numerical approximation of integrals is usually done with quadrature formulas, which have 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 offers `quad` and `quadl` for quadrature in one dimension. These functions are robust and precise, however, they are not very efficient. Both use an adaptive refinement procedure to reduce the number of function calls. However, as `quad` and `quadl` are recursive M-file functions, the algorithmic overhead is significant. Furthermore, adaptive refinement gains little from vectorization.

If an application requires approximating dozens of integrals, and if the integrand function can be efficiently vectorized, using nonadaptive quadrature may improve speed.

```
% Approximate Fourier series coefficients for exp(sin(x)^6) for frequencies -20 to 20
for n = -20:20
   c(n + 21) = quad(inline('exp(sin(x).^6).*exp(-i*x*n)','x','n'),0,pi,1e-4,[],n);
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 enables vectorized computation. Since the integrals are all over the same interval, the nodes and weights need only 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(-i*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(-i*x.*n) );
```

5.1 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 composite 4th-order Gauss-Lobatto [3], based on the approximation

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

```
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 $(\mathtt{N} - 1)/3$ 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 [5, 6]

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

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 Gauss-Lobatto.

For most purposes, composite Simpson's rule is a sufficient default choice. Depending on the integrand, other choices can improve accuracy:

- Use higher-order quadrature formulas if the integrand has many continuous derivatives.
- Use lower-order if the integrand function is not smooth.
- Use the substitution $u = \frac{1}{1-x}$ or Gauss-Laguerre quadrature for infinite integrals like $\int_0^\infty$.

5.2 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`$\times$`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. Nevertheless, the basic Monte Carlo idea is straightforward and of practical value. Suppose that N points, $x_1, x_2, \ldots, x_N$, are uniformly randomly selected in a multidimensional volume Ω. Then

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

To integrate a complicated volume W that is difficult to sample uniformly, find an easier volume Ω that contains W and can be sampled [4]. Then

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

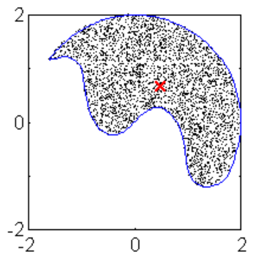
$\chi_W(x)$ is the *indicator function* of W: $\chi_W(x) = 1$ when x is within volume W and $\chi_W(x) = 0$ otherwise. Multiplying the integrand by χ_W sets contributions from outside of W to zero.

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 $M = \int_W \mathrm{d}A$, $M_x = \int_W x\,\mathrm{d}A$, and $M_y = \int_W y\,\mathrm{d}A$, the center of mass is $(\frac{M_x}{M}, \frac{M_y}{M})$. The region is contained in the rectangle Ω defined by $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$. The following code estimates M, M_x, and M_y:

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

%%% Restrict the points to region W %%%
i = find(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 Omega
M = (area/N) * length(x);
Mx = (area/N) * sum(x);
My = (area/N) * sum(y);
```



Region W sampled with $N = 10^4$. The center of mass (the red X) is approximately $(0.47, 0.67)$.

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

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

Monte Carlo integration is a study of its own, see for example [1] for refinements and variations.

6 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.

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 extremal points (local maximums and minimums) of a signal have zero derivative, their locations can be estimated from the zero-crossings of `diff(x)`, provided the signal is sufficiently smooth. For rougher or coarsely sampled signals, a more robust 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;
```

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);
```

FFT-based convolution

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

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

If `x` and `b` are both real, follow this with `y = real(y)`. 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));
```

If `x` and `b` are both real, follow this with `y = real(y)`. If you have the Signal Processing Toolbox, it is faster to use `fftfilt` for FFT-based, zero-padded filtering.

IFFT

As mentioned in section 4 on inlining simple functions, `y = ifft(x)` where `x` is a vector is equivalently and more efficiently implemented with

```
y = conj(fft(conj(x)))/length(x);
```

Though the speed gain is quite small relative to the costly `fft` operation, it is nevertheless an easy additional optimization for the previous two snippets.

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; filter computations involving samples that fall outside of the data 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 $y[n] = b_1 x[n] + b_2 x[n-1]$ (or in Z-transform notation, $b(z) = b_1 + b_2 z^{-1}$) is implemented as

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

Similarly, given a boundary extension value `padRight` for `x(end+1)`, the filter $y[n] = b_1 x[n+1] + b_2 x[n]$ (in Z-transform notation, $b(z) = b_1 z + b_2$) 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 conditions itself requires filtering. In general, to implement noncausal filtering and filtering with other boundary handling methods, it is usually fastest to pad the original 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 `q`, where $1 \leq$ `q0` $\leq$ `q`:

```
y = x(q0:q: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]`.

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.

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.

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

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.

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. For a two-dimensional matrix `A` of size `M` by `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).
```

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` is a vector of rows with `M` elements and `colv` is a vector of columns with `N` elements. 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.

On the right-hand side of an operation (that is, on the right side of the `=`), a vector subscript reference returns a submatrix of elements of size `M`$\times$`N`:

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

If the vector subscripted matrix is on the left-hand side, the right-hand side result must `M`$\times$`N` or scalar size. If any elements in the destination reference are repeated, for example, this ambiguous assignment of `A(1,2)` and `A(2,2)`,

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

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

it is the value in the source array with the greater index that dominates.

```
>> 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.

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 as `indexv`. For example, if `indexv` is a 3×4 matrix, `A(indexv)` is the 3×4 matrix

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

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

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.

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

A =
     0     3
     0     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.

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`.

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 Miscellaneous Tricks

8.1 Clip a value without using if statements

To clip (or clamp, bound, or saturate) a value to within a range, the straightforward way to code this 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, x >= lowerBound
x = min(x,upperBound);      % Clip elements from above, x <= upperBound
```

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

8.2 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 to a column vector if the array is a row vector, a matrix, an N-d array, or already a column vector.

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

By following this operation with a transpose `.'`, it is possible to convert an array to a row vector.

8.3 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. Often it is more useful to find the extreme element of the entire matrix. It is possible to repeat the `min` or `max` function on the column extremes; `min(min(A))` to find the minimum of a two-dimensional matrix `A`. This method uses only one call to `min` or `max` and determines the extreme element's location:

```
[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.)

8.4 Vector Normalization

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

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

The speed gain over a `for` loop is more significant as the number of vectors increases and as the number of dimensions decreases. For several thousand vectors of length 3, the vectorized approach is around 10 times faster.

8.5 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`.

8.6 Vectorized use of set on GUI objects

A serious graphical user interface (GUI) can have dozens of objects like text labels, buttons, and sliders. These objects must all be initialized with the `uicontrol` function with lengthy property names. 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]);
```

This is excessive for just three edit boxes. A vectorized call to `set` can reduce the wordiness:

```
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]);
```

9 Further Reading

For more reading on vectorization, see the MathWorks vectorization guide:
    `http://www.mathworks.com/support/tech-notes/1100/1109.shtml`
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`

JIT Acceleration

MATLAB version 6.5 (R13) and later feature the Just-In-Time (JIT) Accelerator, improving the speed of M-file functions and scripts, particularly self-contained loops. To enable acceleration, code must follow certain guidelines. For details and examples of JIT performance acceleration, see
    `http://www.mathworks.com/access/helpdesk_r13/help/techdoc/matlab_prog/ch7_perf.html`

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 large-scale 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 fast MEX functions that can be called in Matlab just like M-file functions. The typical speed improvement over equivalent M-file 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-file 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-file code. Writing MEX-files requires solid understanding of Matlab and comfort with C or Fortran, and takes more time to develop than M-file code.

Matlab Compiler

The Matlab Compiler generates C and C++ code from M-file functions, producing MEX-files and standalone executables. For product information and documentation, see

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

Good luck and happy coding.

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] W. Gander and W. Gautschi. "Adaptive Quadrature–Revisited," *BIT*, vol. 40, pp. 84-101, 2000.

[4] W. Press, B. Flannery, S. Teukolsky and W. Vetterling. *Numerical Recipes.* Cambridge University Press, 1986.

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

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