Tutorial Overview

- General advice about optimization
- A typical workflow for performance optimization
- MATLAB's performance measurement tools
- Common performance issues in MATLAB and how to solve them
General Advice on Performance Optimization

- "The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet." — Micheal A. Jackson, 1988

- "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified" —- Donald Knuth, 1974

- ...learn to trust your instruments. If you want to know how a program behaves, your best bet is to run it and see what happens" —- Carlos Bueno, 2013
A typical optimization workflow

create
measure
while goals not met
profile
modify
test
measure
end while
A typical optimization workflow

- Design and write the program
- Test to make sure that it works as designed / required
- Don't pay “undue” attention to performance at this stage.
A typical optimization workflow

- Run and time the program
- Be sure to try a typical workload, or a range of workloads if needed.
- Compare your results with your goals/requirements. If it is “fast enough”, you are done!
A typical optimization workflow

- Detailed measurement of execution time, typically line-by-line
- Use these data to identify “hotspots” that you should focus on

create
measure
while goals not met

profile
modify
test
measure
end while
A typical optimization workflow

- Focus on just one “hotspot”
- Diagnose and fix the problem, if you can
A typical optimization workflow

- You just made some changes to a working program, make sure you did not break it!

create
measure
while goals not met
profile
modify
test
measure
end while
A typical optimization workflow

- Run and time the program, as before.

```plaintext
create
measure
while goals not met
    profile
    modify
    test
    measure
end while
```
A typical optimization workflow

- Repeat until your performance goals are met

```plaintext
create
measure
while goals not met
    profile
    modify
    test
    measure
end while
```
Tools to measure performance

- `tic` and `toc`
  - Simple timer functions (CPU time)

- `timeit`
  - Runs/times repeatedly, better estimate of the mean run time, for functions only

- `profile`
  - Detailed analysis of program execution time
  - Measures time (CPU or wall) and much more

- MATLAB Editor
  - Code Analyzer (Mlint) warns of many common issues
Example: sliding window image smoothing

Original: first view of the earth from the moon, NASA Lunar Orbiter 1, 1966

Input: downsampled, with gaussian noise

Output: smoothed with 9x9 window
Where to Find Performance Gains?

- **Serial Performance**
  - Eliminate unnecessary work
  - Improve memory use
  - Vectorize (eliminate loops)
  - Compile (MEX)

- **Parallel Performance**
  - “For-free” in many built-in MATLAB functions
  - Explicit parallel programming using the Parallel computing toolbox
Unnecessary work (1): redundant operations*

Avoid redundant operations in loops:

**bad**

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

**good**

```plaintext
x = 10;
for i=1:N
   .
   .
end
```
Unnecessary work (2): reduce overhead

..from function calls

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

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

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

myfunc2(N);
```

good

..from loops

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

bad

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

bad
Unnecessary work (3): logical tests

Avoid unnecessary logical tests...

...by using short-circuit logical operators

```matlab
if (i == 1 | j == 2) & k == 5
    % do something
end
```

...by moving known cases out of loops

```matlab
for i=1:N
    if i == 1
        % i=1 case
    else
        % i>1 case
    end
end
```

```matlab
if (i == 1 || j == 2) && k == 5
    % do something
end
```

```matlab
for i=2:N
    % i>1 case
end
```
Unnecessary work (4): reorganize equations*

Reorganize equations to use fewer or more efficient operators

Basic operators have different speeds:

Add 3–6 cycles
Multiply 4–8 cycles
Divide 32–45 cycles
Power, etc (worse)

c = 4;
for i=1:N
    x(i)=y(i)/c;
    v(i) = x(i) + x(i)^2 + x(i)^3;
    z(i) = log(x(i)) * log(y(i));
end

s = 1/4;
for i=1:N
    x(i) = y(i)*s;
    v(i) = x(i)*(1+x(i)*(1+x(i)));
    z(i) = log(x(i) + y(i));
end
Unnecessary work (5): avoid re-interpreting code

MATLAB improves performance by interpreting a program only once, unless you tell it to forget that work by running “clear all”

<table>
<thead>
<tr>
<th>Value of ItemType</th>
<th>Variables in scope</th>
<th>Scripts and functions</th>
<th>Class definitions</th>
<th>Persistent variables</th>
<th>MEX functions</th>
<th>Global variables</th>
<th>Import list</th>
<th>Java classes on the dynamic path</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>From command prompt only</td>
</tr>
<tr>
<td>variables</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MATLAB a run faster the 2nd time

Functions are typically faster than scripts (not to mention better in all other ways)
Vectorization is the process of making your code work on array-structured data in parallel, rather than using for-loops.

This can make your code much faster since vectorized operations take advantage of low level optimized routines such as LAPACK or BLAS, and can often utilize multiple system cores.

There are many tools and tricks to vectorize your code, a few important options are:

• Using built-in operators and functions

• Working on subsets of variables by slicing and indexing

• Expanding variable dimensions to match matrix sizes
Memory (1): the memory hierarchy

To use memory efficiently:

- Minimize disk I/O
- Avoid unnecessary memory access
- Make good use of the cache
Arrays are always allocated in contiguous address space.

If an array changes size, and runs out of contiguous space, it must be moved.

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

This can be very very bad for performance when variables become large.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x(1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2000</td>
<td>x(1)</td>
</tr>
<tr>
<td>2001</td>
<td>x(2)</td>
</tr>
<tr>
<td>2002</td>
<td>x(1)</td>
</tr>
<tr>
<td>2003</td>
<td>x(2)</td>
</tr>
<tr>
<td>2004</td>
<td>x(3)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10004</td>
<td>x(1)</td>
</tr>
<tr>
<td>10005</td>
<td>x(2)</td>
</tr>
<tr>
<td>10006</td>
<td>x(3)</td>
</tr>
<tr>
<td>10007</td>
<td>x(4)</td>
</tr>
</tbody>
</table>
Memory (3): preallocate arrays, cont.*

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

\[
A = \text{zeros}(n,m); \quad \% \text{initialize } A \text{ to } 0 \\
A(n,m) = 0; \quad \% \text{or touch largest element}
\]

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

\[
A = \text{rand}(100,100); \\
\% \ldots \\
\% \text{if final size is } 60 \times 40, \text{ remove unused portion} \\
A(61:\text{end,:}) = []; \quad A(:,41:\text{end}) = []; \quad \% \text{delete}
\]
Memory (4): cache and data locality

- Cache is much faster than main memory (RAM)
- Cache hit: required variable is in cache, fast
- Cache miss: required variable not in cache, slower
- Long story short: faster to access contiguous data
Memory (5): cache and data locality, cont.

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

“mini” cache
holds 2 lines, 4 words each

Main memory

<table>
<thead>
<tr>
<th>x(1)</th>
<th>x(9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x(2)</td>
<td>x(10)</td>
</tr>
<tr>
<td>x(3)</td>
<td>a</td>
</tr>
<tr>
<td>x(4)</td>
<td>b</td>
</tr>
<tr>
<td>x(5)</td>
<td>:</td>
</tr>
<tr>
<td>x(6)</td>
<td></td>
</tr>
<tr>
<td>x(7)</td>
<td></td>
</tr>
<tr>
<td>x(8)</td>
<td></td>
</tr>
</tbody>
</table>
Memory (6): cache and data locality, cont.

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

```matlab
for i=1:10
    x(i) = i;
end
```
Memory (7): cache and data locality, cont.

need \(x(5)\), not in cache, cache miss

- load line from memory into cache
- free ride next 3 loop indices, cache hits

```matlab
for i = 1:10
    x(i) = i;
end
```
Memory (8): cache and data locality, cont.

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

```matlab
for i=1:10
    x(i) = i;
end
```
Memory (9): for-loop order*

- Multidimensional arrays are stored in memory along columns (column-major)

- Best if inner-most loop is for array left-most index, etc.

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

good
n=5000; x = zeros(n);
for j = 1:n       % columns
    for i = 1:n    % rows
        x(i,j) = i+(j-1)*n;
    end
end
```
Memory (10): avoid creating unnecessary variables

Avoid time needed to allocate and write data to main memory.

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

\[
\begin{align*}
\text{bad} \\
x &= \text{rand}(5000); \\
y &= x.^2;
\end{align*}
\]

\[
\begin{align*}
\text{good} \\
x &= \text{rand}(5000); \\
x &= x.^2;
\end{align*}
\]

Caveat: May not be work if the data type or size changes – these changes can force reallocation or disable JIT acceleration.