TUNING MATLAB FOR BETTER PERFORMANCE

Keith Ma
Boston University
Research Computing Services

BOSTON UNIVERSITY

Tutorial Overview

- General advice about optimization
- A typical workflow for performance optimization
- MATLAB's performance measurement tools
- Common performance issues in MATLAB
- Worked example: image smoothing (moving average)

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

```
create
measure
while goals not met
    profile
    modify
    test
    measure
end while
```

- 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

```
create
measure
while goals not met
    profile
    modify
    test
    measure
end while
```

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

A typical optimization workflow

```
create
measure
while goals not met
    profile
    modify
    test
    measure
end while
```

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

A typical optimization workflow

```
create
measure
while goals not met
    profile
    modify
    test
    measure
end while
```

- Focus on just one "hotspot"
- Diagnose and fix the problem, if you can

A typical optimization workflow

```
create
measure
while goals not met
    profile
    modify
    test
    measure
end while
```

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

A typical optimization workflow

```
create
measure
while goals not met
    profile
    modify
    test
    measure
end while
```

- Run and time the program, as before.

A typical optimization workflow

```
create
measure
while goals not met
    profile
    modify
    test
    measure
end while
```

- Repeat until your performance goals are met

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

Where to Find Performance Gains ?

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

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

good

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

Unnecessary work (2): reduce overhead

..from function calls

bad

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

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

good

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

myfunc2(N);
```

..from loops

bad

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

good

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

Unnecessary work (3): logical tests

Avoid unnecessary logical tests...

...by using short-circuit logical operators

bad

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

good

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

...by moving known cases out of loops

bad

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

good

```
% i=1 case
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)
```

bad

```
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
```

good

```
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): don't 'clear all'

Value of `ItemType`	Items Cleared							
	Variables in scope	Scripts and functions	Class definitions	Persistent variables	MEX functions	Global variables	Import list	Java classes on the dynamic path
`all`	✓	✓		✓	✓	✓	From command prompt only	
`classes`	✓	✓	✓	✓	✓	✓	✓	
`functions`		✓		✓	✓			
`global`						✓		
`import`							✓	
`java`	✓	✓		✓	✓	✓		✓
`mex`					✓			
`variables`	✓							

MATLAB improves performance by interpreting a program only once, unless you tell it to forget that work

Vectorize (1)

MATLAB is designed for vector and matrix operations. The use of *for*-loop, in general, can be expensive, especially if the loop count is large and nested.

When possible, use vector representation instead of *for*-loops.

bad

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

good

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

Vectorize (2): why is is faster?

- Implicit (automated, internal) parallelization
- Highly-tuned, compiled, math libraries employing state-of-the-art algorithms and performance optimizations techniques
- Highly-tuned, compiled, core MATLAB functions
- Make use of CPU-level vectorization

Vectorize (3): example using logical arrays

Logical arrays can be used for indexing:

bad

```
d = rand(1000, 1);
h = rand(1000, 1);

v = [ ];
for n = 1:1000
    if h(n) > 0.5
        v(end+1) =
1/12*pi*(d(n)^2)*h(n));
    end
end
```

good

```
d = rand(1000, 1);
h = rand(1000, 1);

mask = h<0.5;
v = 1/12*pi*(d(mask)^2)*h(mask));
```

Vectorize (4): example using logical arrays

Or in other, more creative, ways:

bad

```
A = rand(100,1);
B = rand(100,1);

for i = 1:100
   if B(i)>0.5
      C(i) = A(i)^2;
   else
      C(i) = exp(B(i));
   end
end
```

good

```
A = rand(100,1);
B = rand(100,1);

D = (B>0.5);
C = D.*(A.^2)+(~D).*exp(B);
```

Vectorize (5): example using repmat

repmat helps construct the matrices needed for vectorized calculations

```
x = -3:0.01:3;
y = -3:0.01:3;

xx = repmat(x, numel(y),        1);
yy = repmat(y' ,        1, numel(x));

plane = 5+3*xx+2*yy;
```

Vectorize (6): example using bsxfun

bsxfun provides a way of combining matrices of different dimensions without using repmat to match their size first

```
A = [97 89 84; 95 82 92; 64 80 99];
Abar = mean(A);

dev = A - repmat(Abar, size(A,1), 1);
```

```
A = [97 89 84; 95 82 92; 64 80 99];
Abar = mean(A);

dev = bsxfun(@minus, A, Abar);
```

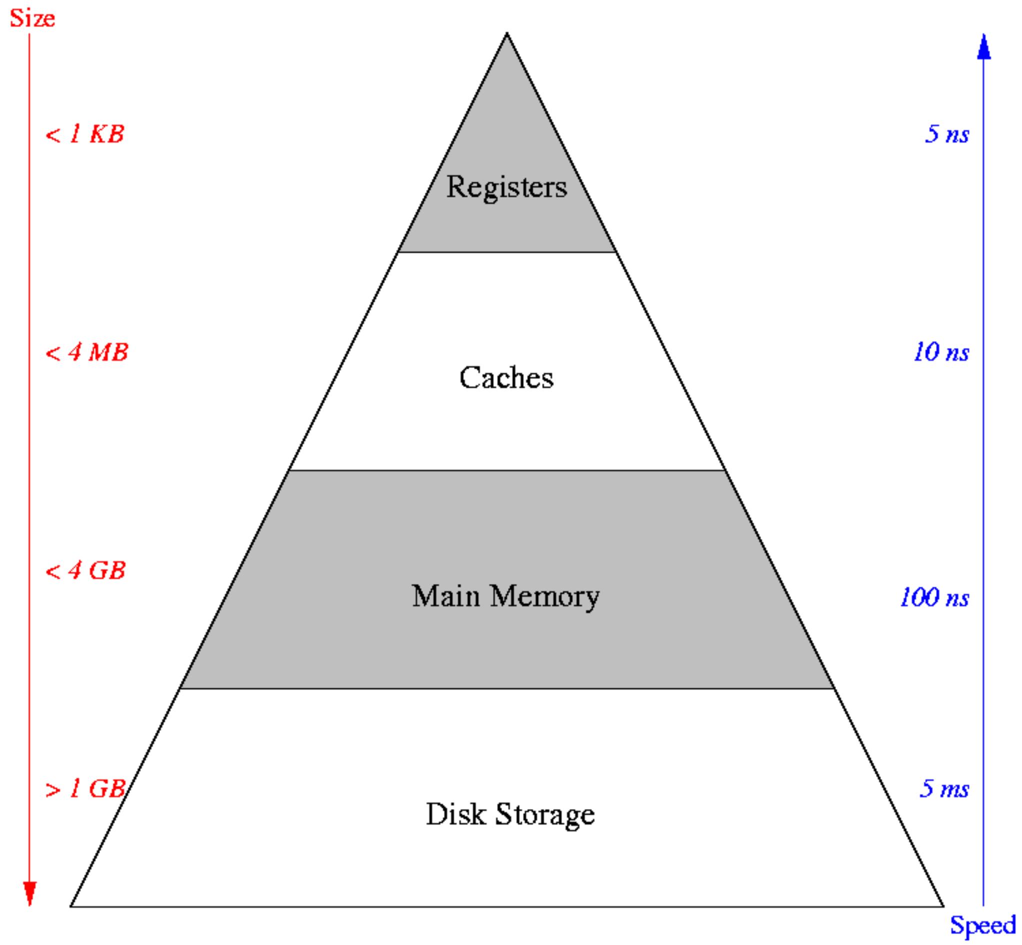
Valid operations are:

@plus, @minus, @times, @rdivide, @ldivide, @power, @max, @min, @rem, @mod, @atan2, @atan2d, @hypot, @eq, @ne, @lt, @le, @gt, @ge, @and, @or, @xor

Vectorize (6): other utility functions

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

Memory (1): the memory hierarchy



To use memory efficiently:

- Minimize disk I/O
- Avoid unnecessary memory access
- Make good use of the cache

Memory (2): preallocate arrays

- 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

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

Memory (3): preallocate arrays, cont.

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

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

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

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

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.

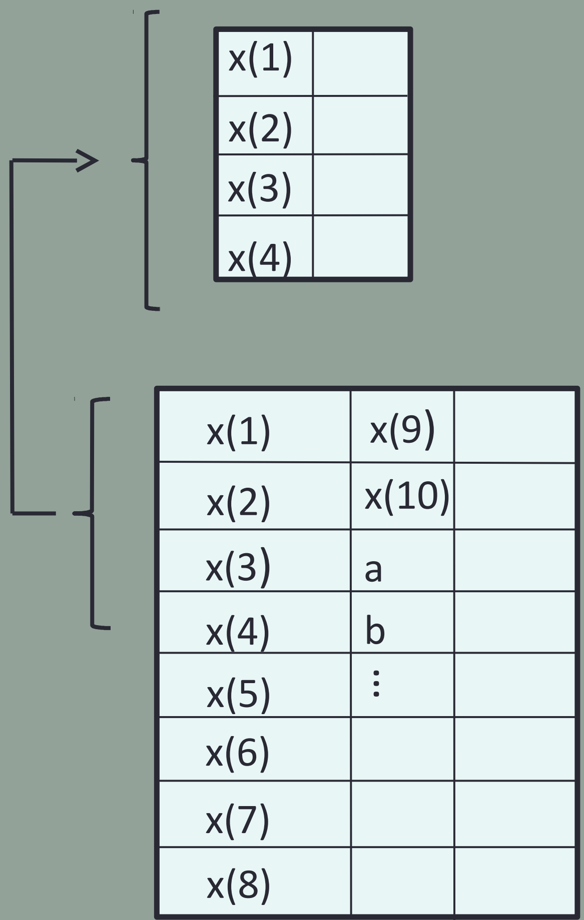
"mini" cache
holds 2 lines, 4 words each

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

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

Main memory

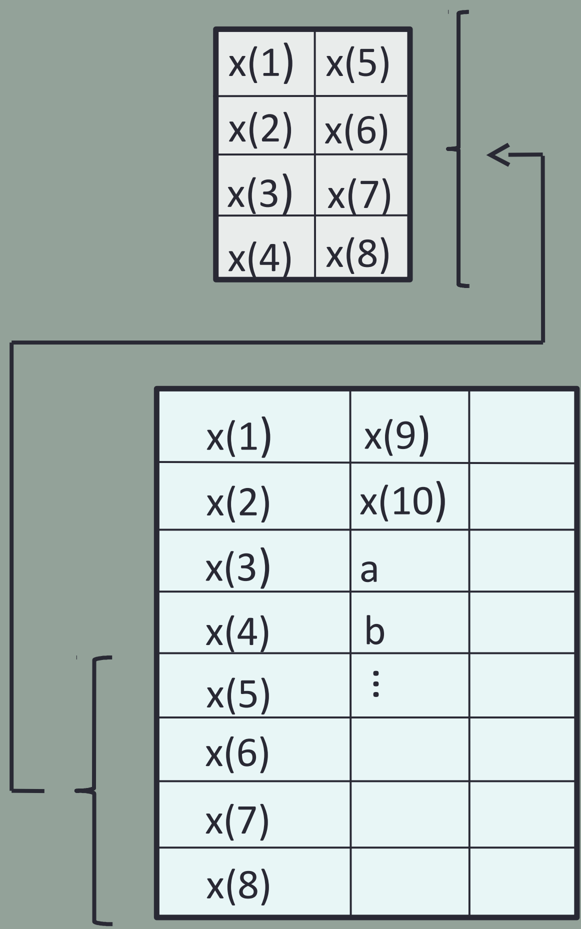
Memory (6): cache and data locality, cont.



- 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

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

Memory (7): cache and data locality, cont.

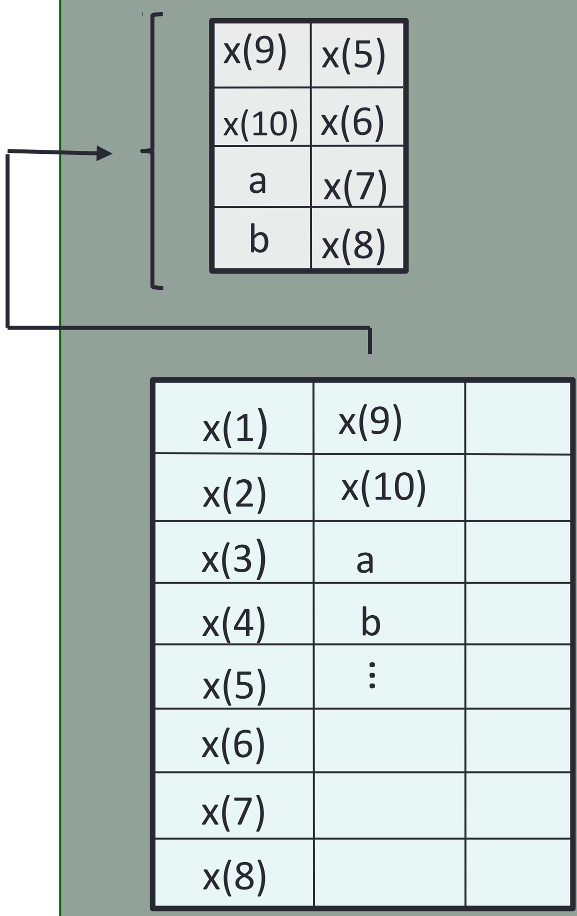


need x(5), not in cache, cache miss

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

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

Memory (8): cache and data locality, cont.



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

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

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): compute-in-place

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

bad

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

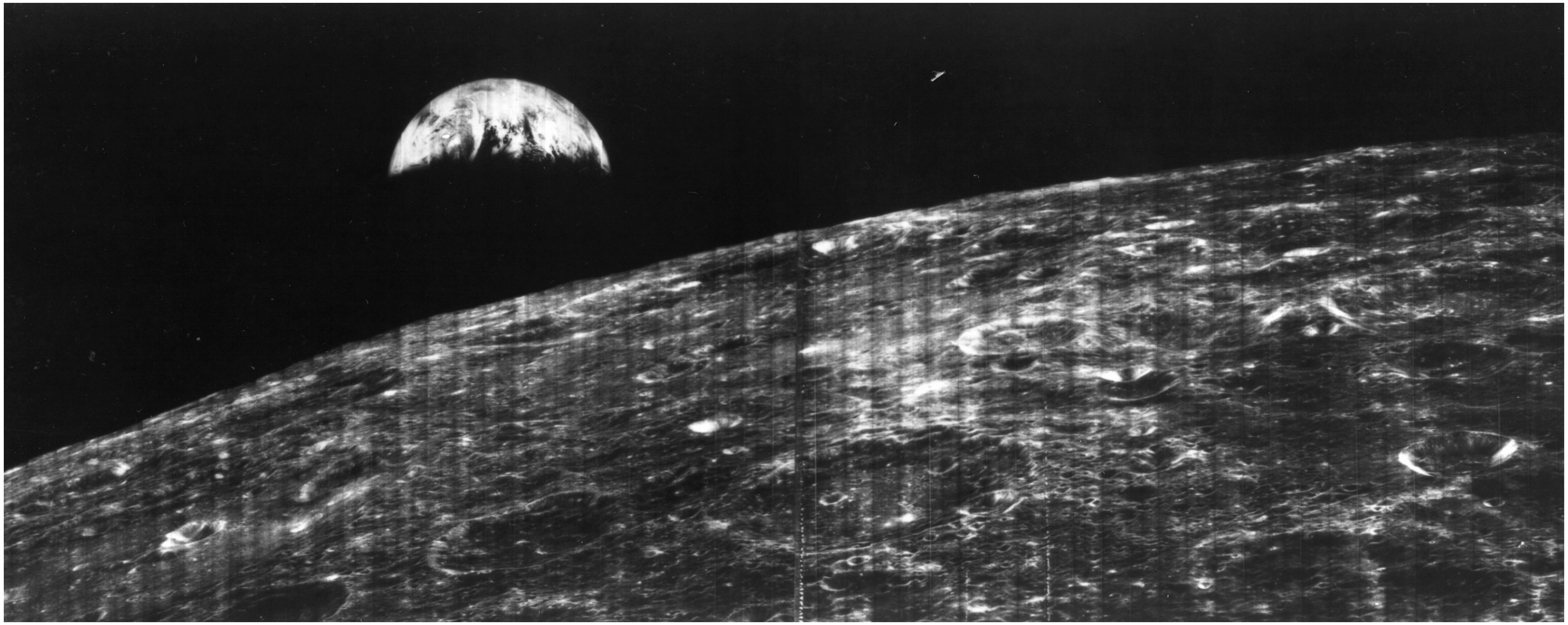
good

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

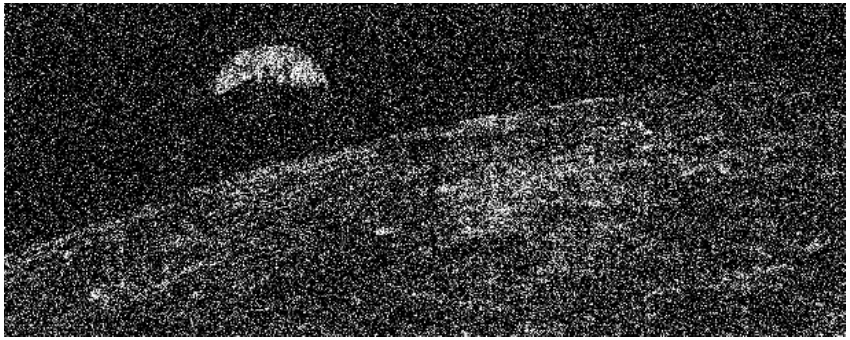
Caveat: May not be work if the data type or size changes – these changes can force reallocation or disable JIT acceleration

More generally, avoid temporary variables

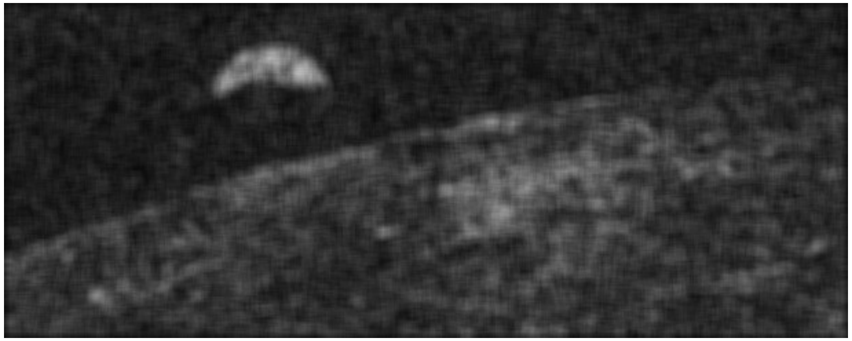
Worked Example: sliding window image smoothing



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



Input: downsampled, with gaussian noise



Output: smoothed with 9x9 window