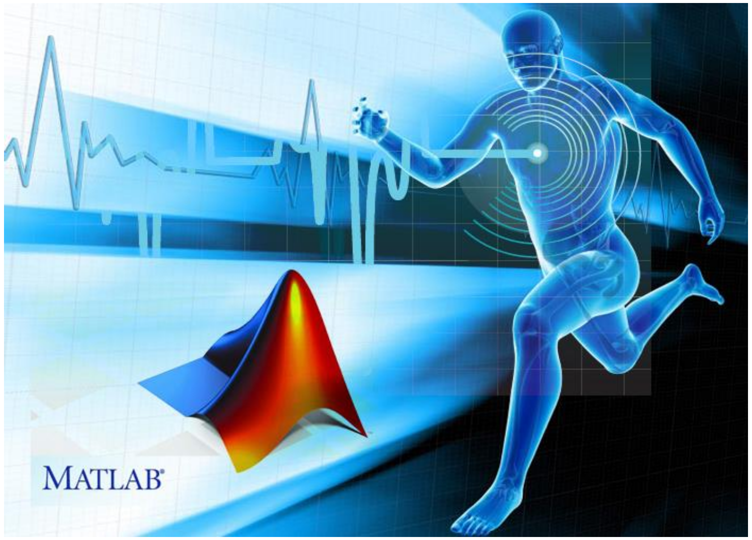
Improve MATLAB Code Quality and Performance

Shiran Golan

Application Engineer

Shirang@systematics.co.il



Outline

- Improving Code Quality
- Improving Code Performance
- Summary

Outline

- Improving Code Quality
- Improving Code Performance
- Summary

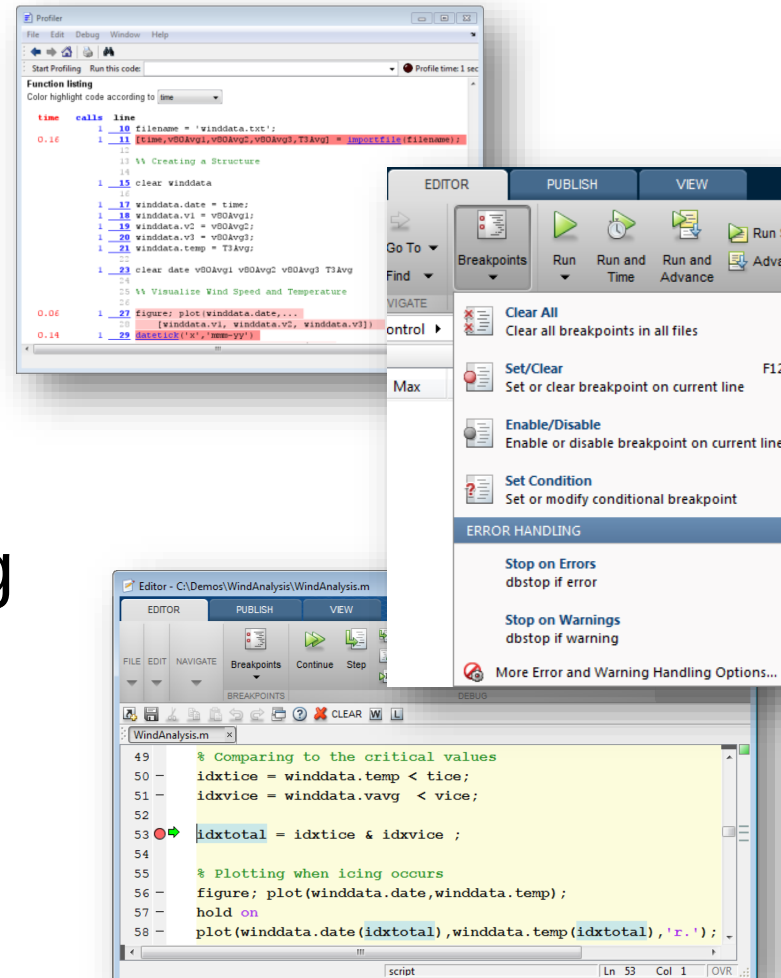
Code Quality

- Writing “better” code
  - Less error-prone
  - Human readable code
  - Performance tuning

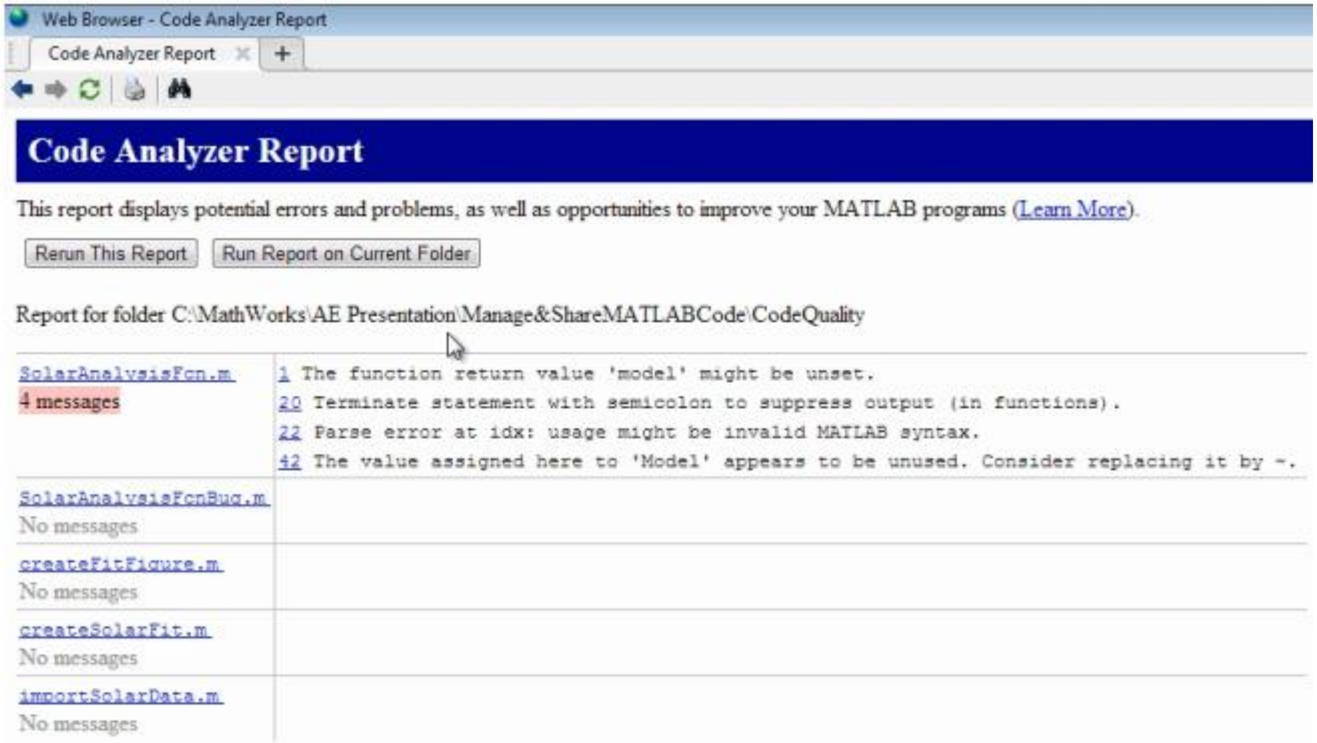
- Robustness
  - Validate, guard inputs/outputs
  - Handle errors, exceptions

Improving Code Quality in MATLAB

- Code Analyzer
- McCabe complexity
- Input and error handling
- Debugging



MATLAB Code Analyzer (previously known as mlint)



- Run `checkcode` on multiple files in folder and generate a report

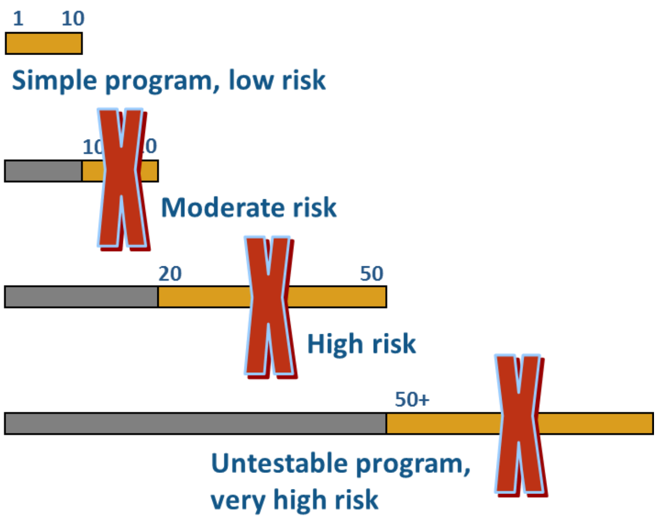
Check McCabe Complexity

- McCabe complexity (`checkcode -cyc`)
  - Quantitative measure of the complexity of a program

Lower complexity	**➔**	**Easier to understand, modify**
Higher complexity	**➔**	**More likely to contain errors**

- Can lower the complexity by dividing a function into smaller, simpler functions

- Good rule of thumb is to aim for complexity around 10 or lower



Input and Error Handling

- Check input arguments
  - `validateattributes`
  - `inputParser`
- Handle exceptions
  - `try ... catch`
  - `MException`
- Warn or Error
  - `assert`
  - `warning, error`

```
try
    dist = fitdist(returns, distName);
catch ME
    if strcmp(ME.identifier, 'stats:ProbDistUnivParam:che
        error('parametricVaR:unrecognizedDistribution', '
    else
        rethrow(ME);
    end
end
```

Outline

- Improving Code Quality
- Improving Code Performance
- Summary

Code Performance

- Power of vector & matrix operations

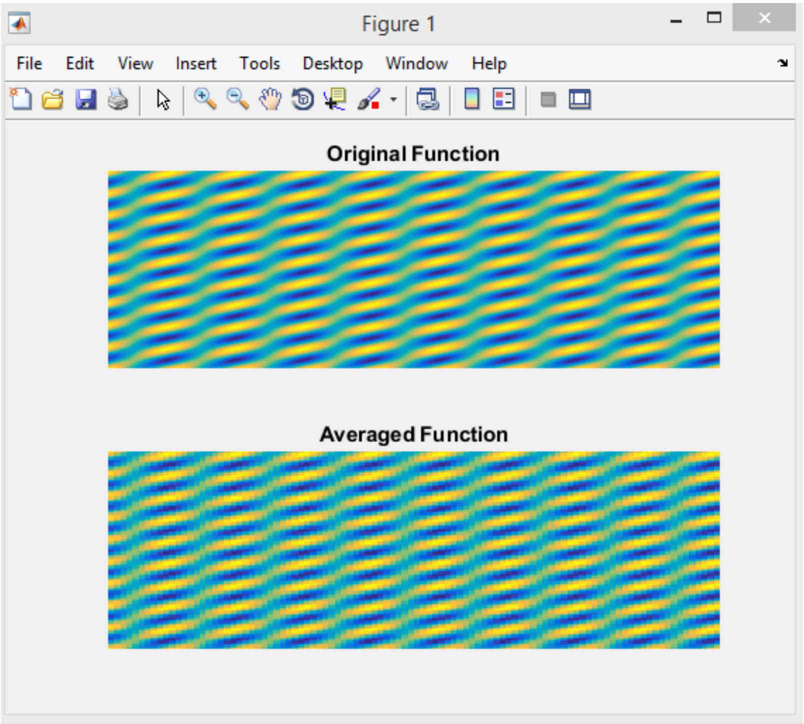
  Example: Block Processing Images

- Addressing bottlenecks

  Example: Fitting Data

Example: Block Processing Images

- Evaluate function at grid points
- Reevaluate function over larger blocks using averaging
- Compare the results

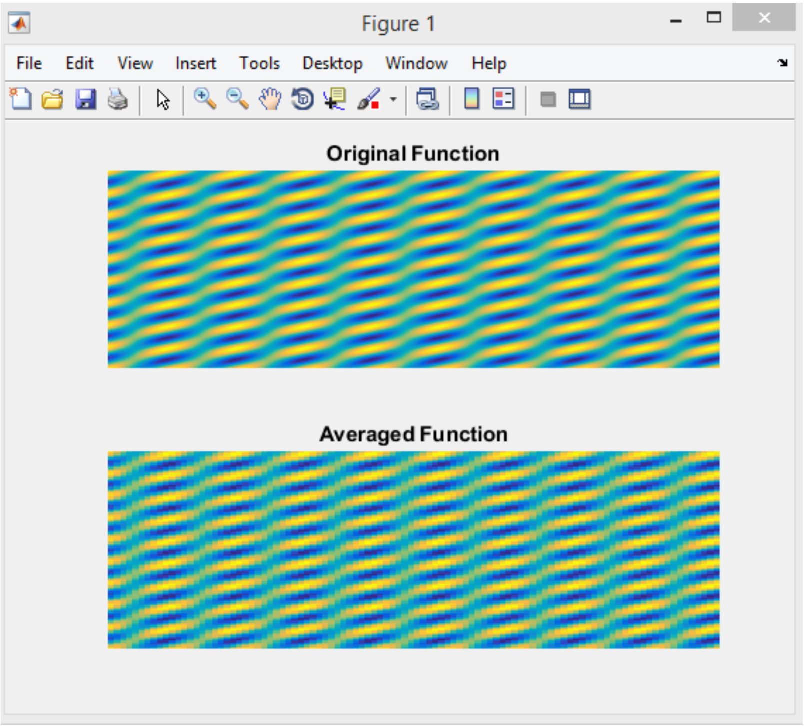


Summary of Example

- Used built-in timing functions

```
>> tic
>> toc
```

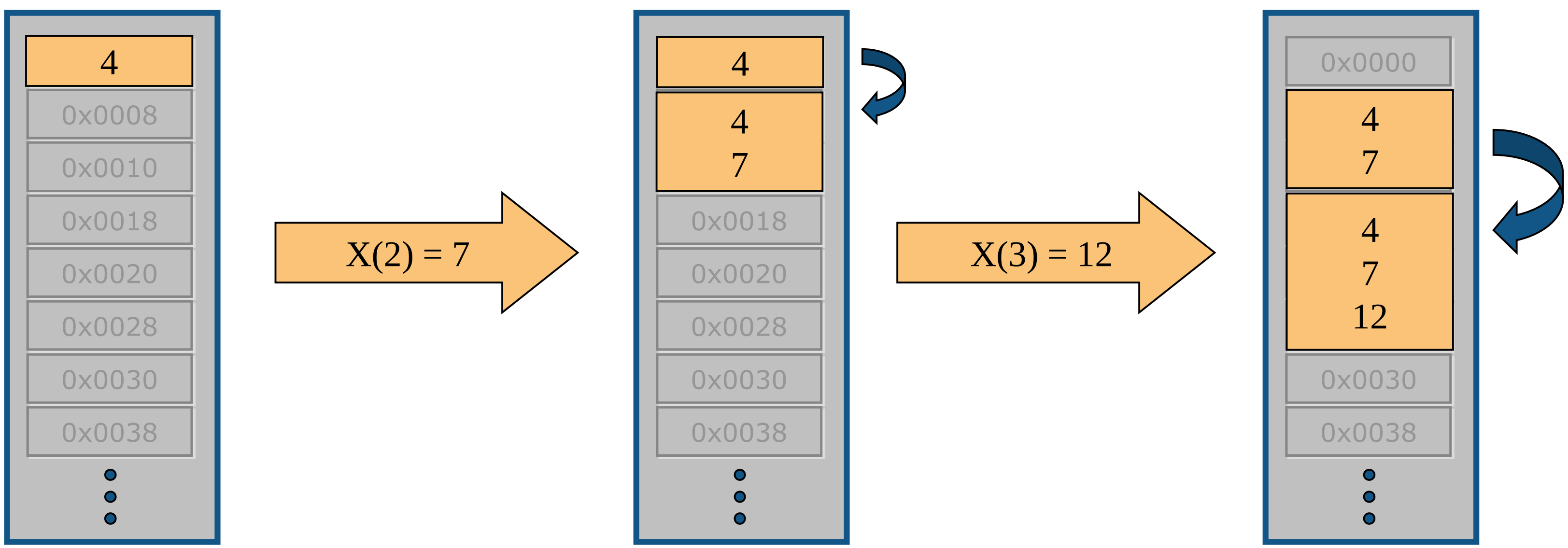
- Used Code Analyzer to find suboptimal code
- Preallocated arrays
- Vectorized code



Effect of Not Preallocating Memory

```
>> x = 4
>> x(2) = 7
>> x(3) = 12
```

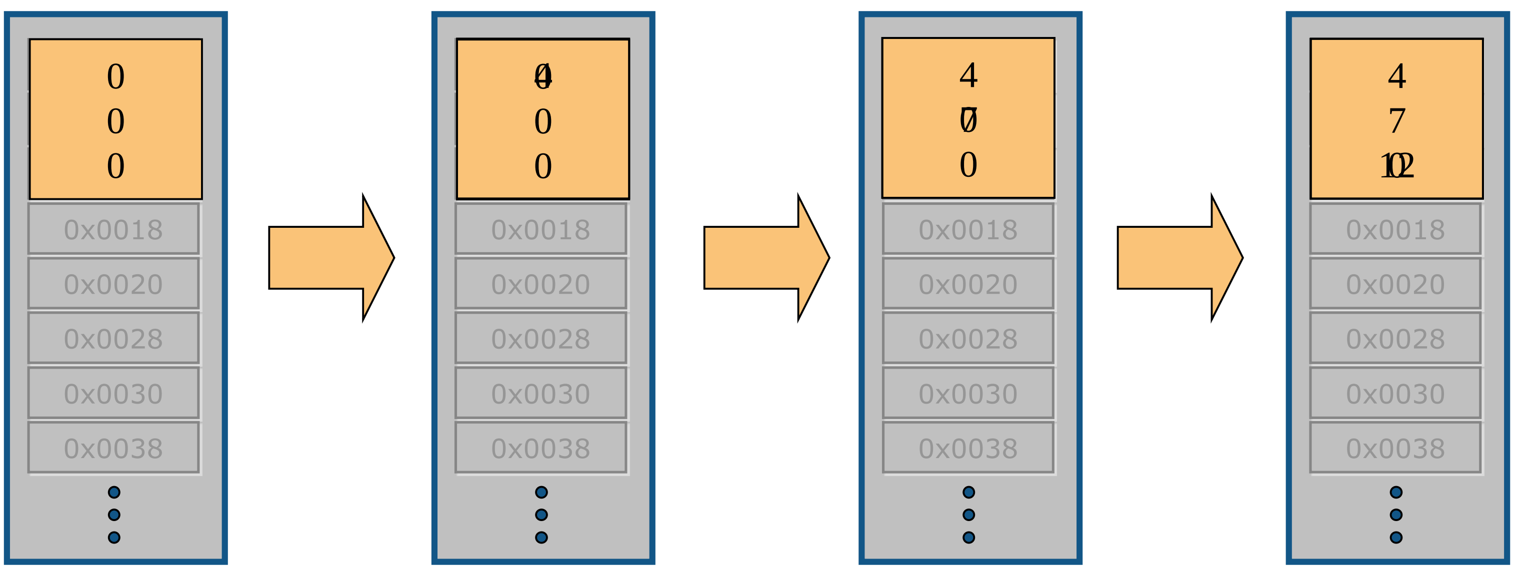
Resizing Arrays is Expensive



Benefit of Preallocation

Reduced Memory Operations

```
>> x = zeros(3,1)
>> x(1) = 4
>> x(2) = 7
>> x(3) = 12
```



Speed and Memory Usage

- Balance **vectorization** and **memory usage**
  - Use `implicit expansion` instead of functions such as `repmat`
  - Reduce size of arrays to smaller blocks for block processing
- Consider using sparse matrices
  - ***Less Memory***: Store only nonzero elements and their indices
  - ***Faster***: Eliminate operations on zero elements

Implicit Expansion

R2016b

Automatic expansion of element-wise operations

- Replacing **`bsxfun`** function, or **`repmat`** expansion
- No more `Matrix dimensions must agree`
- Advantages:
  - Faster
  - Better memory usage
  - Improved readability of code

```
a = [3 2 3;
     4 2 4;
     5 2 2]

b = mean(a)

res = a-b
```

```
a =

     3     2     3
     4     2     4
     5     2     2

b =

     4     2     3

res =

    -1     0     0
     0     0     1
     1     0    -1
```

Implicit Expansion

R2016b

Support for implicit expansion:

- Element-wise arithmetic operators:

`+ , - , .* , .^ , ./ , .\`

- Relational operators: `<, <=, >, >=, ==, ~=`
- Logical operators: `&, |, xor`
- Bit-wise functions: `bitand, bitor, bitxor`
- Elementary math functions:

`max, min, mod, rem, hypot, atan2, atan2d`

Sparse Matrices

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

- Why use sparse?
  - ✓ **Less Memory**
    Store only the nonzero elements of the matrix and their indices
  - ✓ **Faster**
    Reduce computation time by eliminating operations on zero elements
- When to use sparse?
  - ✓ < 1/2 dense (on 64-bit, double precision)

Using Sparse Matrices

- **Functions**
  That support sparse matrices
  `>> help sparfun`

- **Creation**
  `S = sparse(i,j,v,m,n)`
  `A = spdiags(B,d,m,n)`

```
i = [900 1000];
j = [900 1000];
v = [10 100];
S = sparse(i,j,v,1500,1500)
```

```
S =

   (900,900)      10
  (1000,1000)    100
```

```
size(S)
```

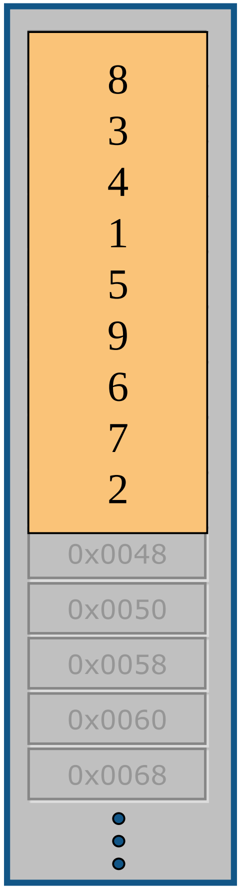
```
ans =

        1500        1500
```

- **Blog Post: Creating Sparse Finite Element Matrices**
  http://blogs.mathworks.com/loren/2007/03/01/creating-sparse-finite-element-matrices-in-matlab/

Data Storage of MATLAB Arrays

```
>> x = magic(3)
x =
      8     1     6
      3     5     7
      4     9     2
```

Column-Major Memory Storage



- Iterate over columns
- Use columns first in nested loop
- MATLAB functions work on columns by default:

  `>> sum(x)` for columns

  `>> sum(x,2)` for rows

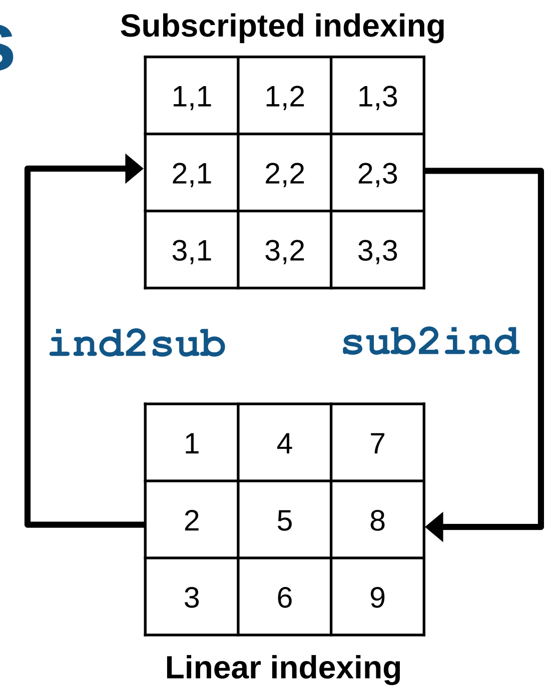
Use Only the Precision You Need

- Numerical data types
  – Float: double and single precision (8 and 4 bytes)
  – Integer: signed and unsigned (1-4 bytes)

- Floating point for math (e.g. linear algebra)

- Integers where appropriate (e.g. images)

Indexing into MATLAB Arrays



- Subscripted
  - Access elements by rows and columns
- Linear
  - Access elements with a single number
- Logical
  - Access elements with logical operations or mask

Indexing - Summary

- Indexing:

Linear Indexing is **faster** than Subscripted Indexing

- Conditional Indexing:

Logical indexing — **Fast**

`find` function

Nested for loop — **Slow**

MATLAB Underlying Technologies

- Commercial libraries
  - BLAS: Basic Linear Algebra Subroutines (multithreaded)
  - LAPACK: Linear Algebra Package
  - etc.

BLAS and LAPACK require contiguous arrays

MATLAB Underlying Technologies

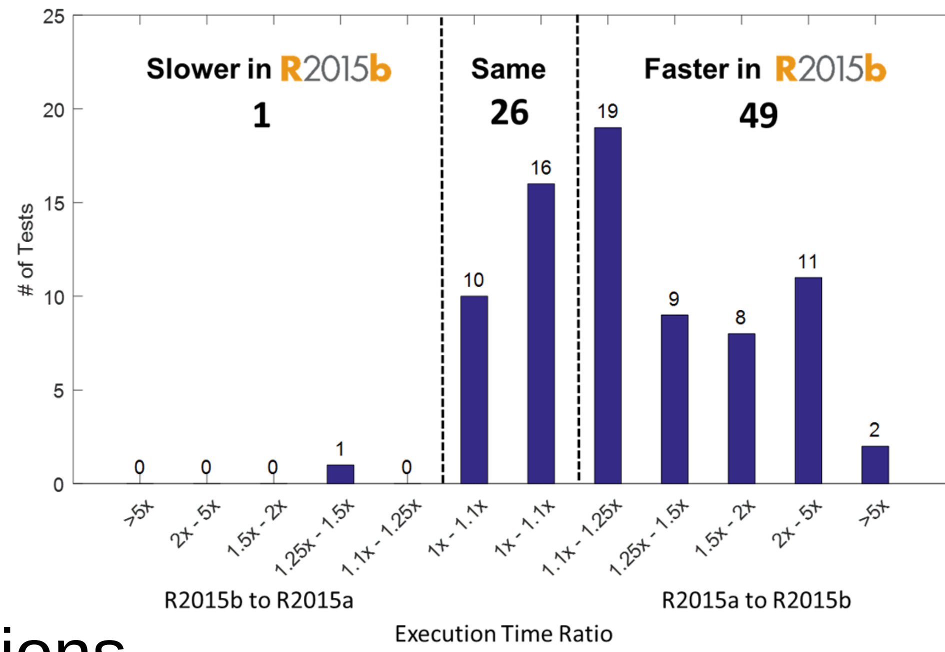
JIT(Just-in-time) compilation:

- Redesigned in **2015b:**

  - Function calls

  - Object-oriented features

  - Element-wise math operations



- Continually improving – **2016b**:

  - Tight loops execution

  - Objects construction

Code Performance

- Power of vector & matrix operations

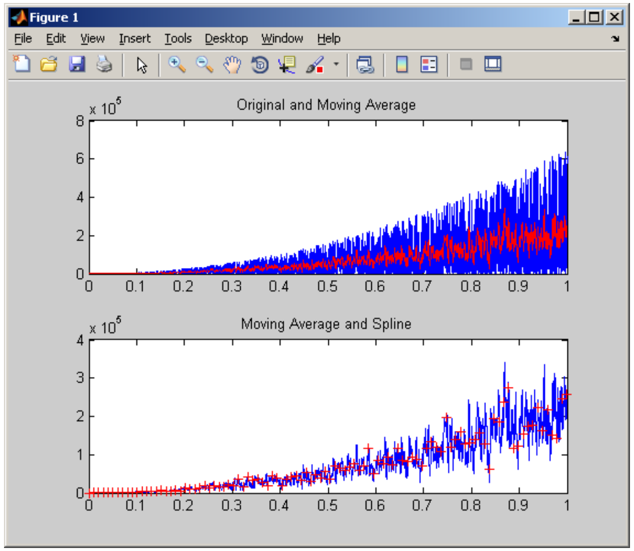
  Example: Block Processing Images

Addressing bottlenecks

  Example: Fitting Data

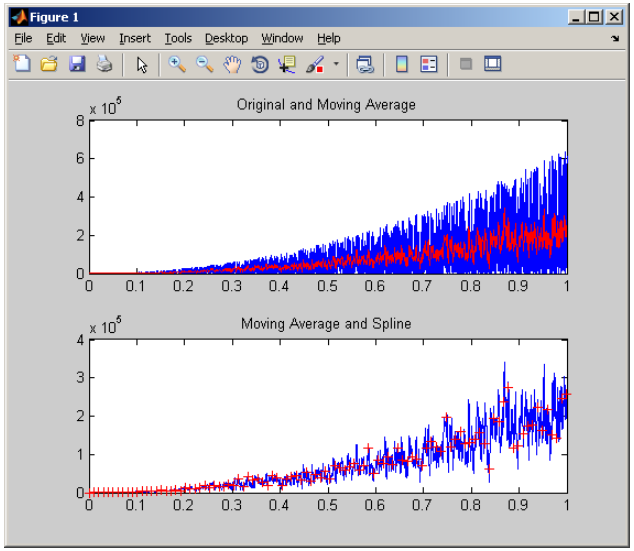
Example: Fitting Data

- Load data from multiple files
- Extract a specific test
- Fit a spline to the data
- Write results to Microsoft Excel



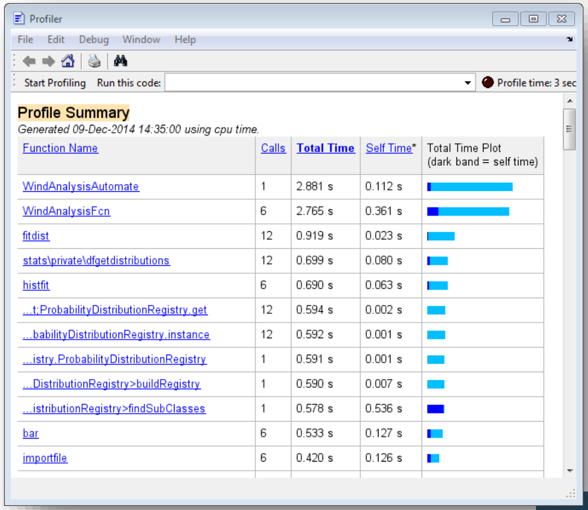
Summary of Example

- Used profiler to analyze code
- Targeted significant bottlenecks
- Reduced file I/O
- Reused figure



Measuring Code Performance

- `tic` and `toc`
  - For timing for smaller portions of code and scripts
  - Measures performance using a **stopwatch timer**
- `timeit`
  - For timing a function
  - Measures the function multiple times and computes the median
- Profiler
  - For identifying specific performance bottlenecks in code
  - Measures relative execution time



Interpreting Profiler Results

- Focus on **top bottleneck**
  - Total number of function calls
  - Time per function call
- Functions
  - All function calls have **overhead**
  - Find the **right** function – performance may vary
    - Search MATLAB functions (e.g. `textscan` or `dlmread`)
    - Many shipping functions have viewable source code (`edit` *function_name)*
    - Write a custom function (dedicated functions may be faster)

Code Performance -Summary

Techniques for addressing performance:

- Vectorization
- Preallocation
- Sparse matrices
- Subscripted vs. linear vs. logical

Techniques for addressing bottlenecks:

- Measuring tools
- Reduce File I/O
- Reduce displaying outputs

Outline

- Improving Code Quality
- Improving Code Performance
- Summary

Steps for Improving Performance

- First focus on getting your code working
- Get the latest MATLAB release
- Then speed up the code within core MATLAB
- Consider additional processing power

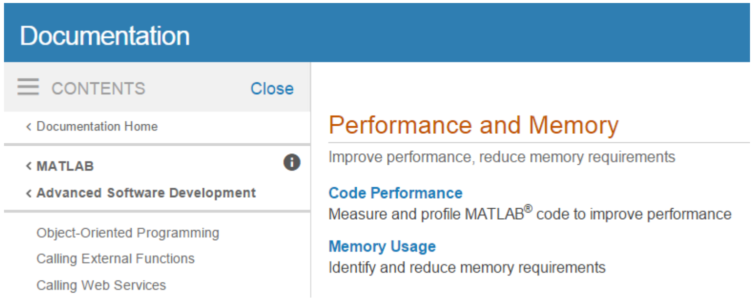
Key Takeaways

- Use MATLAB tools to create a readable and robust code
- Consider performance benefit of vector and matrix operations in MATLAB
- Analyze your code for bottlenecks and address most critical items
- **Take advantage of additional computing resources**

Other Performance Resources

- MATLAB documentation

MATLAB → Advanced Software Development → Performance and Memory



- The Art of MATLAB, Loren Shure's blog

  blogs.mathworks.com/loren/

- Systematics Courses

  http://www.systematics.co.il/courses/mathworks/