Maximum MATLAB

John Burkardt
Department of Scientific Computing
Florida State University

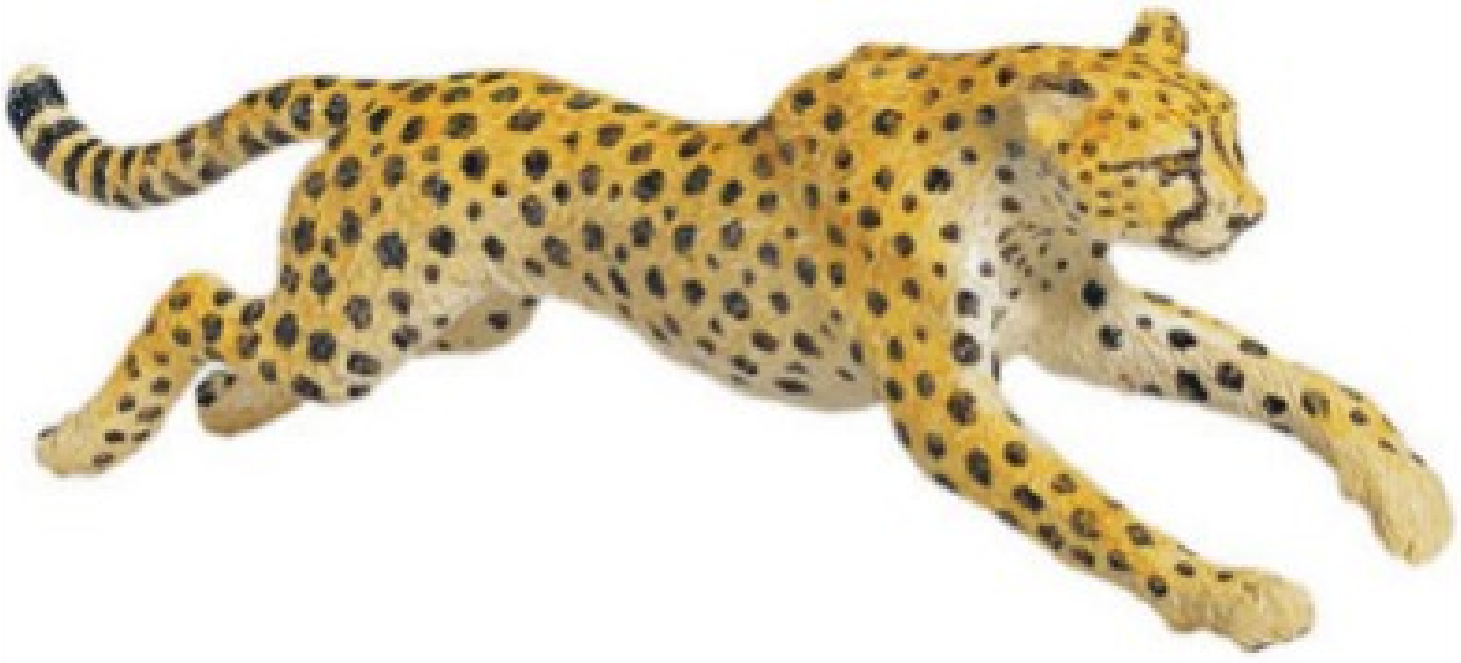
.....

Presentation to the Mathematics Graduate Students
Auburn University
https://people.sc.fsu.edu/~jburkardt/presentations/...
matlab_fast_2012_auburn.pdf

22 March 2012

Introduction

With MATLAB, we can write programs fast.



But can we write fast programs?

Introduction

MATLAB has much of the power of traditional programming languages such as C/C++ and FORTRAN.

But it simplifies or skips many of the features of such languages that can slow down a programmer.

In particular, MATLAB:

- doesn't make you declare your variables;
- doesn't need to compile your program;
- includes a powerful library of numerical functions;
- can be used to edit, debug, run, visualize;
- is easy to use interactively.

Introduction

MATLAB is interactive, but has been written so efficiently that many calculations are carried out as fast as (and sometimes faster than) corresponding work in a compiled language.

So MATLAB can be a comfortable environment for the serious programmer, whether the task is small or large.

However, it's not unusual to encounter a MATLAB program which mysteriously runs very very slowly. This behavior is especially puzzling in cases where the corresponding C or FORTRAN program shows no such slowdown.

You can stop using MATLAB, or stop solving big problems... but sometimes an investigation will get you back on track.

Introduction

Often, the underlying problem can be detected, diagnosed, and corrected, resulting in an efficient MATLAB program.

We will look at some sensible ways to judge whether a MATLAB program is running efficiently, try to guess the "maximum speed" possible for such a program, and consider what to expect for running time when a MATLAB program is given a series of tasks of increasing size.

Once we know what to expect, we'll pick some examples of simple operations that seem to suffer from a slowdown, and try to spot what's wrong and fix it.

Introduction

We will find that MATLAB's editor is one source of helpful warnings and advice for creating better programs.

We will also see that a performance analyzer can watch our program execute and give us an idea of where the most computations are being carried out - those are the places that really need to be made efficient.

We will also look at how MATLAB performance can be improved when the calculation can be written in terms of matrix and vector operations.

Maximum MATLAB

1. **TIC/TOC**
2. What's the Speed Limit?
3. Making Space
4. Using a Grid
5. Sparse Matrices

TIC/TOC: We Are Baffled When MATLAB is Slow

MATLAB executes our commands as fast as we can type them in...

...until it doesn't!

TIC/TOC: Bad Performance Suggests Bad Coding

For casual computations, we don't really care if we have to wait a second or two, and this means we pick up some bad coding habits.

Unfortunately, when we try to solve larger problems, writing bad MATLAB code will make it impossible to solve problems that are actually well within MATLAB's capability.

The way you code a problem can make a big difference.

When performance is an issue, you need to understand:

- how the program's work load grows with problem size;
- how fast your program is running;
- how fast your program ought to run.

TIC/TOC: Can We Quantify Fast Performance?

To call a problem "big", we need to be able to measure the work the computer has been asked to do.

To call an algorithm, computation, or computer "fast", we need to be able to measure time.

If we can make these measurements, we can generalize the formula

```
Speed = Distance / Time,
```

to define

```
Computer Performance = Work / Time
```

Then we can estimate the "speed limit" on our computer, and whether a MATLAB program is performing well or poorly.

TIC/TOC: Time is Measured by TIC and TOC

MATLAB's **tic** function starts or restarts a timer;
toc prints the elapsed time in seconds since **tic** was called.

```
tic
a = rand(1000,1000);
toc
```

If we type these lines interactively, then the timer is also measuring the speed at which we type! For quick operations, the typing time exceeds the computational time.

TIC/TOC: An Interactive Example of Timing

```
>> tic
>> a = rand ( 1000, 1000 );
>> toc
Elapsed time is 7.625469 seconds.
>> tic
>> a = rand ( 1000, 1000 );
>> toc
Elapsed time is 14.400338 seconds.
>> tic
>> a = rand ( 1000, 1000 );
>> toc
Elapsed time is 10.408739 seconds.
>>
```

But I don't want to know my bad (and variable) typing speed!

TIC/TOC: A Noninteractive Timing Example

We can avoid the typing delay by putting our commands into a MATLAB M file, called **ticker.m**, which repeats the timing 5 times:

```
>> ticker
Elapsed time is 0.015173 seconds.
Elapsed time is 0.021067 seconds.
Elapsed time is 0.016615 seconds.
Elapsed time is 0.015537 seconds.
Elapsed time is 0.015265 seconds.
>>
```

These times are much smaller and less variable than the interactive tests. Even here, though, we see that repeating the exact same operation doesn't guarantee the exact same timing.

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/ticker.m

TIC/TOC: Measuring Work is Difficult

Measuring the work involved in a computer program is harder than measuring time. A single MATLAB statement can represent almost any amount of computation.

And because MATLAB is *interpreted*, a program like this:

Statement1.
Statement2.

is really something like this:

Have MATLAB interpret Statement1 and set up for it.
Execute **Statement1.**
Have MATLAB interpret Statement2 and set up for it.
Execute **Statement2.**

Time spent interpreting and setting up is time **not spent** on your computation!

Maximum MATLAB

1. TIC/TOC
2. **What's the Speed Limit?**
3. Making Space
4. Using a Grid
5. Sparse Matrices

LIMIT: What's the Fastest We Can Go?

You don't know what fast is

...until you can't possibly go any faster!

LIMIT: Your Computer Reports a Clock Rate

The built-in information about my computer reports:

```
2.8 Ghz Quad-Core Intel Xeon
```

This is a rating for the clock speed. We can think of it as meaning that the computer's heart beats 2.8 billion times per second. Or perhaps we should call it the brain, instead.

It's generally true that any operation on a computer will take at least one clock cycle. So no matter what action we want to do, we can't do more than 2.8 billion of them in a second. Which doesn't really seem like a very difficult limit to live with!

But do I have any idea whether my programs run that fast?

LIMIT: A Dot Product is a Simple Computation

Let's test what the 2.8 billion "speed limit" means.

Given two vectors $\vec{u}$ and $\vec{v}$, the scalar dot product is defined by:

$$s = \vec{u}^T \vec{v} = \sum_{i=1}^{N} u_i v_i$$

and can be computed in about **4*N** operations:

- 1 initialization of **s**
- 2***N** "fetches" from memory of u_i and v_i
- **N** multiplies of $u_i * v_i$
- **N** additions of $s + u_i * v_i$
- 1 write to memory of s

We'll assume we can do just one operation per clock cycle.

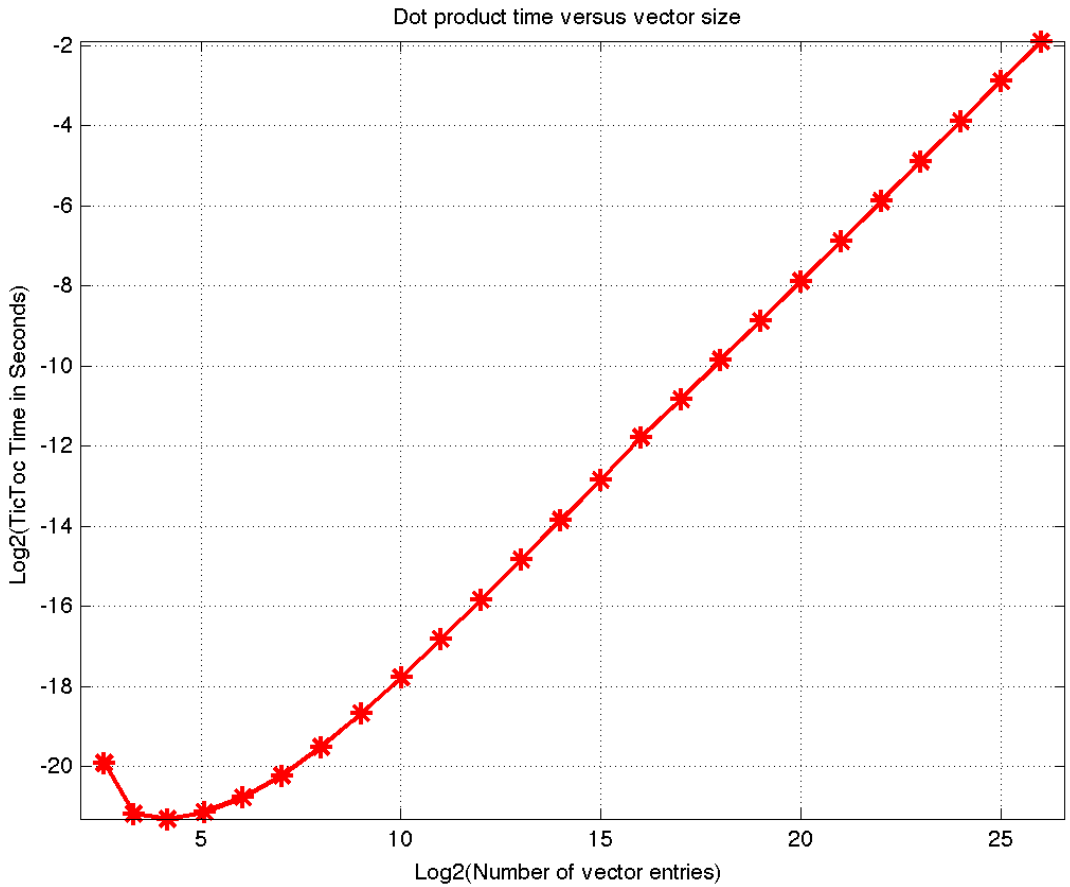
LIMIT: Compute Dot Products with a FOR Loop

```
x = rand ( n, 1 );
y = rand ( n, 1 );

tic;
z = 0.0;
for i = 1 : n
  z = z + x(i) * y(i);
end
t = toc;

speed = (4*n+2) / t;
speed_limit = 2800000000;

fprintf ( '%d  %g  %g\n', n, speed, speed_limit );
```

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/dot_for_graph.m

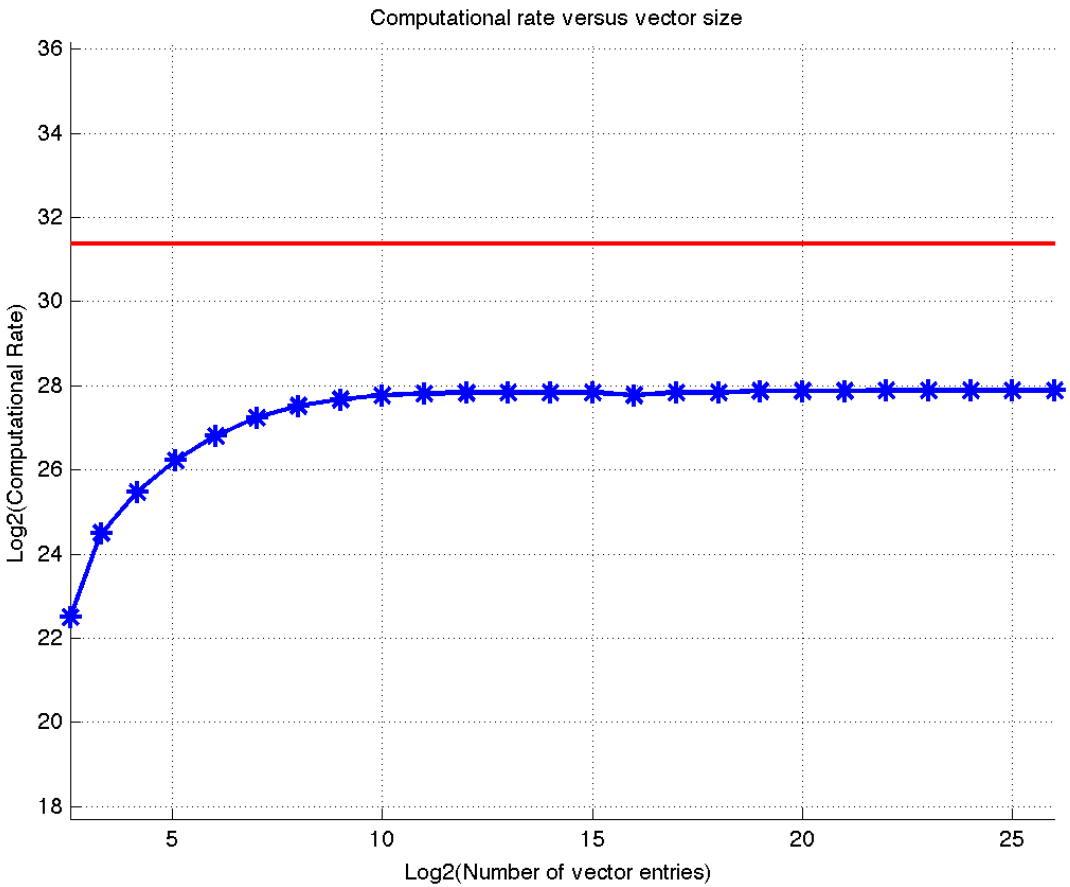
LIMIT: The Time Increases Linearly With Work

Nice linear growth (if we can ignore the peculiar beginning!)



LIMIT: But Our Speed is a Bit Disappointing

The red line is the "speed limit". We're using logarithms base 2, so our blue line is actually about 10 times slower than the red "speed limit"!



LIMIT: Run MATLAB Program = MATLAB + Program

It's natural to ask what the computer is doing for 9 out of every 10 clock cycles, since it's not working on our problem!

Recall that MATLAB is an interpreted language. That means that when you run a program, you're actually running MATLAB and MATLAB is running your program. So how fast things happen depends on the ratio of MATLAB action and program action.

It turns out that MATLAB has a fair amount of overhead in running a **for** loop. It's not easy to explain the ratio of 9/10, but you can imagine MATLAB setting the index, checking the index, determining which vector entries to retrieve, and so on.

This is like a sandwich with a lot of bread, and not much meat!

LIMIT: Recompute, Using Unrolled Loop

To convince you that the problem is too much MATLAB and not enough computation, let's make a fatter sandwich by putting more "meat" in each execution of the loop:

```
tic;
 z = 0.0;
 for i = 1 : 8 : n
   z = z + x(i)   * y(i)   + x(i+1) * y(i+1) ...
         + x(i+2) * y(i+2) + x(i+3) * y(i+3) ...
         + x(i+4) * y(i+4) + x(i+5) * y(i+5) ...
         + x(i+6) * y(i+6) + x(i+7) * y(i+7);
 end
 t = toc;
```

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/dot_forplus_graph.m

LIMIT: More "Meat" Runs Faster

The exact same computation runs 50% faster now, because MATLAB spends less time setting up each iteration.

Log2(N)	Thin Rate	Fat Rate
20	2.5e+08	3.8e+08
21	2.5e+08	3.7e+08
22	2.5e+08	3.8e+08
23	2.5e+08	3.8e+08
24	2.5e+08	3.8e+08

```
Maximum rate assumed to be 2.8e+09
```

LIMIT: Reformulate Using MATLAB Vectors

When MATLAB is given a loop to control, some overhead occurs because MATLAB doesn't know what is going on inside the loop.

Our loop is a simple vector operation, and if MATLAB knew that, it could coordinate the operations much better, so that the memory reads, arithmetic computations, and memory writes are going on simultaneously.

MATLAB recognizes vector operations if we use the vector notation. In particular, the dot product of two column vectors is expressed by

```
z = x' * y;
```

Let's see if MATLAB takes advantage of the vector formulation.

LIMIT: Recompute Using Vectors

```
x = rand ( n, 1 );
y = rand ( n, 1 );

tic;
z = x' * y;
t = toc;

speed = (4*n+2) / t;
speed_limit = 2800000000;

fprintf ( '%d  %g  %g\n', n, speed, speed_limit );
```

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/dot_vector_graph.m

LIMIT: Did We Go Faster Than Light?

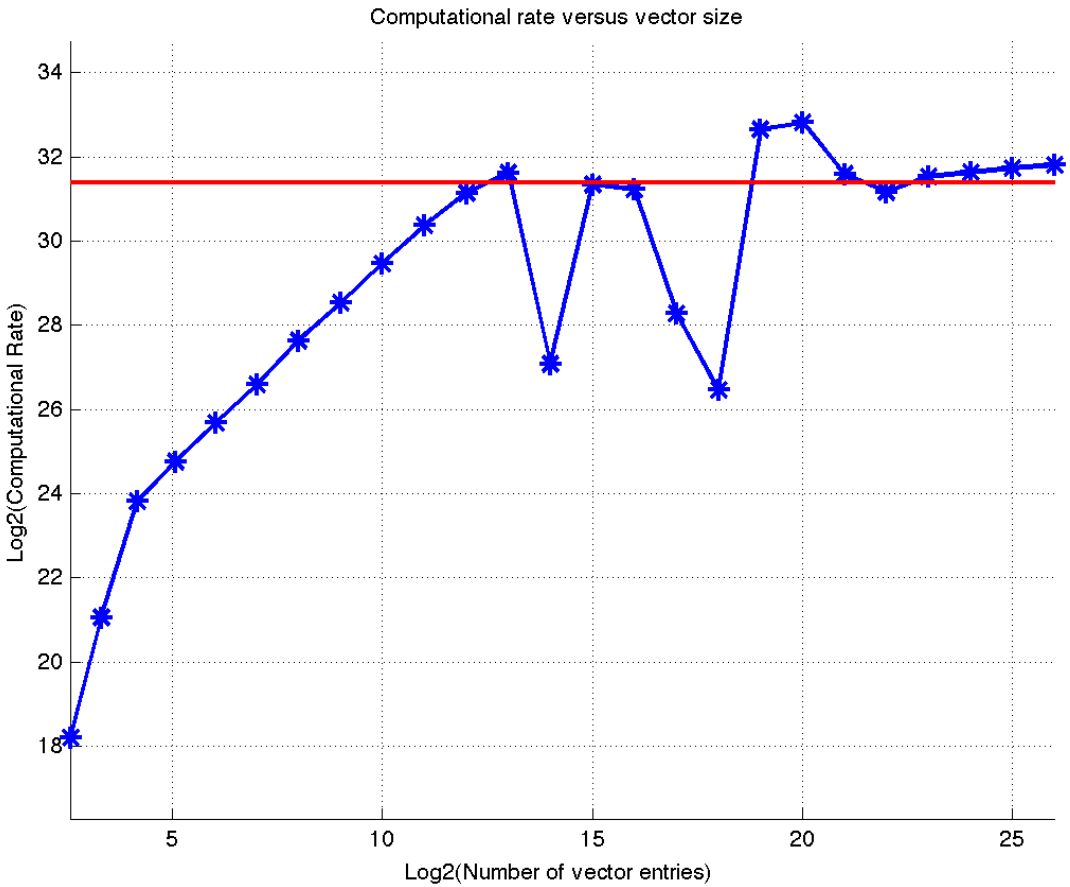
Using vectors pushes the rate to the limit...and beyond!

Log2(N)	Thin Rate	Fat Rate	Vector Rate
20	2.5e+08	3.8e+08	2.4e+09
21	2.5e+08	3.7e+08	3.1e+09
22	2.5e+08	3.8e+08	3.3e+09
23	2.5e+08	3.8e+08	3.6e+09
24	2.5e+08	3.8e+08	3.7e+09

```
Maximum rate assumed to be 2.8e+09
```

We can only assume that, knowing it's a vector operation, MATLAB is able to organize the calculation in a way that beats the "one operation per clock cycle" limit on a typical computer.

LIMIT: Vector Rates Jump Up



LIMIT: Remarks

Now we know what the "speed limit" means.

In a simple computation, where we can count the operations, we can estimate our program's speed and compare it to the limit.

And now we realize that the overhead of running MATLAB can sometimes outweigh our actual computation, especially for loops with only a few operations in them.

Performance might be enhanced by "fattening" such a loop.

Some small loops can be rewritten as vector operations, which can achieve high performance.

Do you prefer OCTAVE? Compare the performance of OCTAVE and MATLAB on the **for** *and vector versions of the dot product.*

Maximum MATLAB

1. TIC/TOC
2. What's the Speed Limit?
3. **Making Space**
4. Using a Grid
5. Sparse Matrices

SPACE: Automatic Storage is Convenient but Hazardous

MATLAB automatically sets aside space for our data as we go...

...but the results can be chaotic!

SPACE: Calculate Array Entries

Let's define a matrix **A** using a formula for its elements. A pair of **for** loops run through the values of **i** and **j**:

```
for i = 1 : m
  for j = 1 : n
    a(i,j) = sin ( i * pi / m ) * exp ( j * pi / n );
  end
end
```

This seems the logical way to define the matrix, and for small **m** and **n**, there's little to say.

But we make two bad MATLAB programming decisions here that will cost us dearly if we look at larger versions of **A**.

---

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/array1.m

SPACE: The Work is Proportional to Matrix Size

To estimate the performance of this calculation, we ought to know how much work is involved in evaluating the formula. But **sin()** and **exp()** are not simple floating point operations, so we can't count the work that way. However, let's simply assume that computing each entry of the matrix costs the same work **W**. In that case, the total work in evaluating the whole matrix is

```
Work = M * N * W
```

So a matrix with 100 times as many elements has 100 times as much work, and presumably takes 100 times the time.

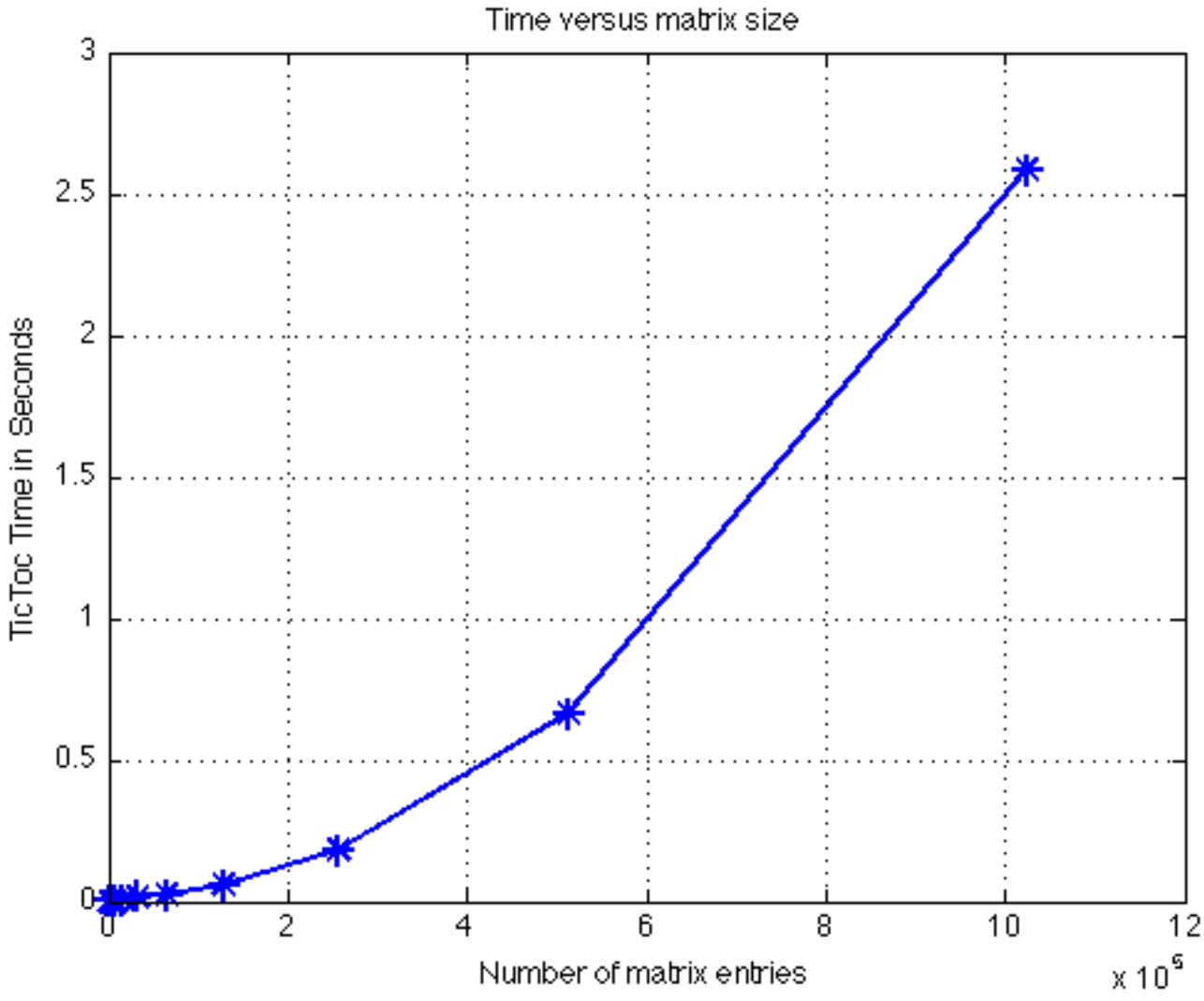
It's not hard to check this using a graph!

SPACE: Time the Problem at Various Sizes

```
m = 1000;
n = 1;
for logn = 0 : 10
  tic;
  for i = 1 : m
    for j = 1 : n
      a(i,j) = sin ( i * pi / m ) * exp ( j * pi /n );
    end
  end
  x(logn+1) = m * n;
  y(logn+1) = toc;
  n = n * 2;
end
plot ( x, y, 'b-*', 'LineWidth', 2, ...
  'MarkerSize', 10 );
```

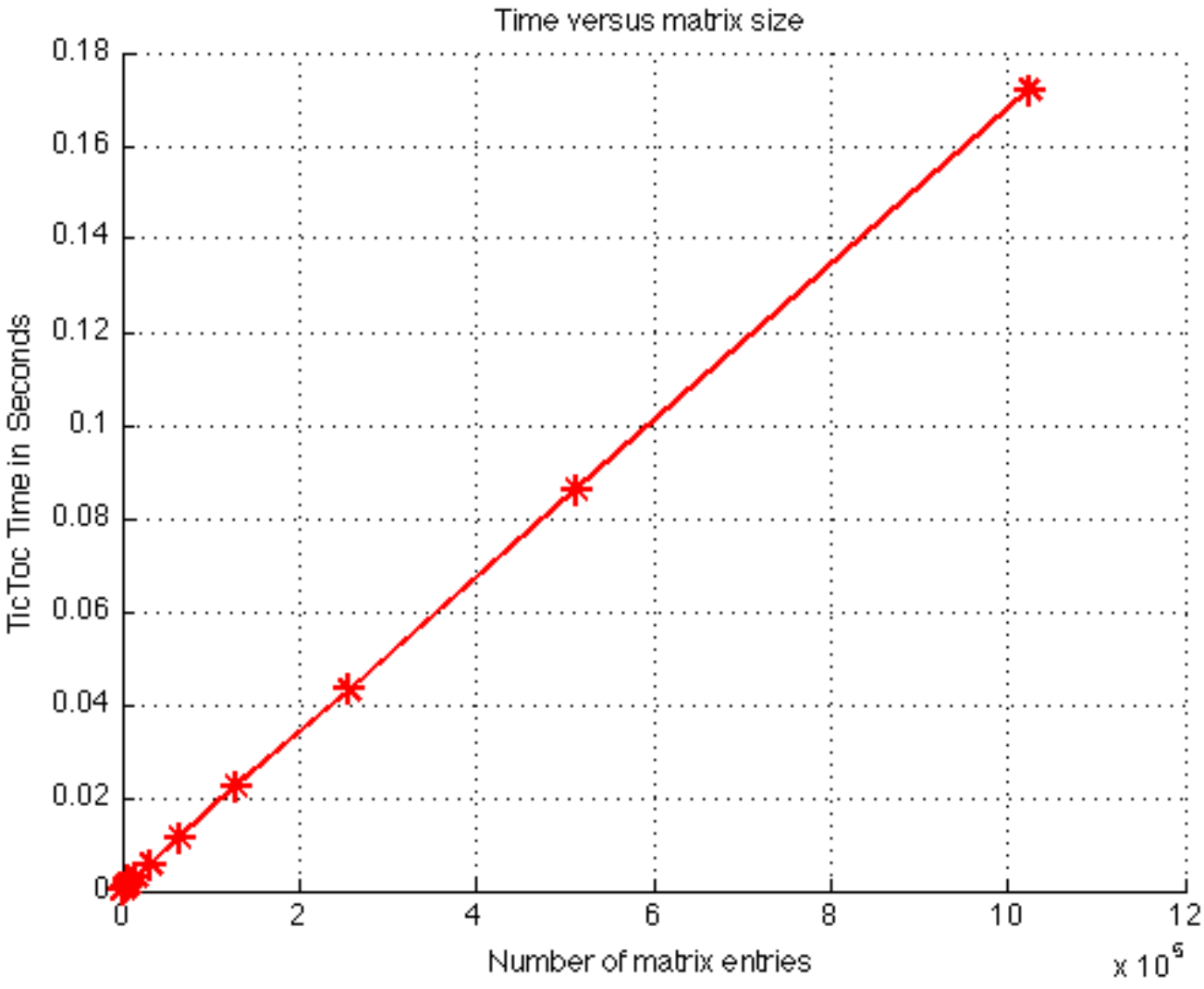
http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/array1_once.m

SPACE: Timing Data for ARRAY1_ONCE



SPACE: Timing Data for Second Run of ARRAY1_ONCE

What happens if we run the program again, right away?



SPACE: How Different Are the Two Runs?

The program **array1_twice.m** runs the computation twice, plotting in blue the first time, and red the second.

(And it uses the **clear** command at the beginning, so we have a clean start!)

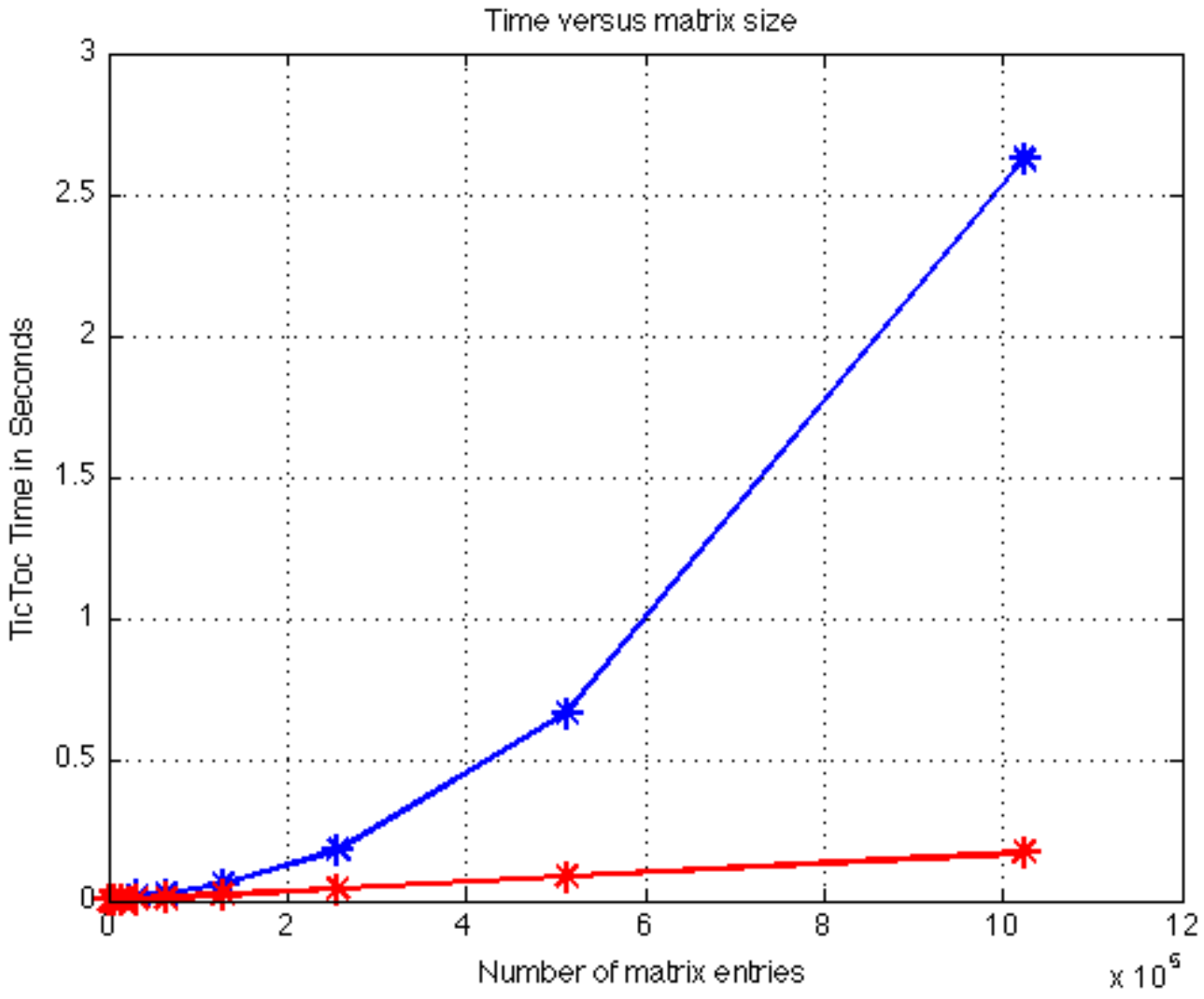
Perhaps if we plot the data together we can understand why the shape of the plot changed.

---

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/array1_twice.m

SPACE: Compare Timings for First and Second Call

The blue line (first call) is actually a quadratic.
The red line (second call) is the linear behavior we expected.



SPACE: Call ARRAY1 Several Times

Look at a similar experiment, for M = N = 1000:

```
                M       N      Time      Rate
              ----    ----  --------  --------
m = 1000
n = 1000
array1        1000    1000  3.167540    315,702
array1        1000    1000  0.189113  5,287,850
array1        1000    1000  0.188934  5,292,870
array1        1000    1000  0.189222  5,284,810
array1        1000    1000  0.188770  5,297,460
clear
m = 1000
n = 1000
array1        1000    1000  3.162318    316,224
array1        1000    1000  0.190403  5,252,030
```

SPACE: Implicitly Declared Arrays Are Expensive

If you use arrays, and don't declare their size, MATLAB cleverly makes sure there is enough space.

MATLAB does this implicitly. If it sees a reference to **x(8)**, it checks if **x** exists, and if not, it creates it.

It checks if **x** has at least 8 entries, and if not, it gets 8 elements of computer memory and copies the old **x** to this new space.

What happens if you have a **for** loop in which you assign entry **i** of a array **x** that was never referenced before?

MATLAB allocates 1 entry and computes it.
Then it allocates 2 entries, copies 1, and computes the last.
Then it allocates 3 entries, copies 2, and computes the last.
...
Creating an array of size 1000 will involve 1000 separate allocations, and the copying of 999*1000/2 entries!

SPACE: Preallocate Arrays!

The entire problem disappears if you simply warn MATLAB in advance that you are going to need a given amount of space for an array. The typical procedure to do this is the **zeros(m,n)** command:

```
a = zeros ( 1000, 1000 );
```

When we ran **array1** a second time, the array space was already allocated, so we only had the computational time to worry about.

SPACE: Even MATLAB's Editor Knows This

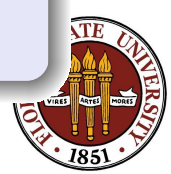
MATLAB's editor can spot and warn you about some inefficiencies like this.

If you use the editor to view a program, on the right hand margin of the window you will see a small red, orange or green box at the top, and possible orange or red tick marks further down, opposite lines of the program.

If you examine **array1.m** this way, you might see an orange box and an orange tick mark. Putting the mouse on the tick mark brings the following message:

Note:

The variable 'a' appears to change size on every loop iteration (within a script). Consider preallocating for speed.



SPACE: Avoid Repeated Computations of Sine and EXP

The code calls **sin()** and **exp()** a total of **2*m*n** times. These are relatively expensive calls, and we could get our results with just **m** calls to **sin()** and **n** calls to **exp()**, at the cost of a little memory.

```
for i = 1 : m
  u(i) = sin ( i * pi / m );
end
for j = 1 : n
  v(j) = exp ( j * pi / n );
end
for i = 1 : m
  for j = 1 : n
    a(i,j) = u(i) * v(j)
  end
end
```

SPACE: There's Always Room For Vectors!

But more importantly, now we can see that we can use MATLAB's vector notation to define **u**, **v** and **a**.

```
u = sin ( ( 1 : m ) * pi / m ); <-- (ROW vector)
v = exp ( ( 1 : n ) * pi / n ); <-- (Row vector)
a = u' * v;                     <-- (Matrix, not scalar!)
```

u and **v** are 1xM and 1xN row vectors, so that **u**' * **v** is an (Mx1)x(1xN) = (MxN) array.

This notation gives MATLAB as much information as we can to help it speed up the execution of these operations.

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/array1_vector.m

SPACE: FOR Loops Lose to Vectors

Let's compare our **for** loop based calculation against the MATLAB vector calculation:

M	N	Rate1	Rate2
1000	1	5.2e+06	8e+05
1000	2	6.7e+06	3.5e+07
1000	4	7.8e+06	7.1e+07
1000	8	8.2e+06	1e+08
1000	16	8.5e+06	1.4e+08
1000	32	8.9e+06	1.6e+08
1000	64	9.3e+06	1.5e+08
1000	128	9.3e+06	1.8e+08
1000	256	9.3e+06	1.7e+08
1000	512	9.3e+06	1.5e+08
1000	1024	4.6e+06	1.4e+08

The improvement is on the order of a factor of 15 or more!

SPACE: Estimating the Improvement Factors

We looked at three ways to compute the entries of the matrix A.

When the matrix is of size about 1000 by 1000, the three ways have very different performance:

Method	Seconds	Rate
Simple	3.22	320,000
Allocate array	0.24	4,300,000
Use Vectors	0.0071	140,000,000

so the first improvement produced a factor of 10 speedup, and the second a factor of 30.

Our rates are in "results per second", since we don't know how to measure the work involved in computing **sin()** and **exp()**.

SPACE: Remarks

In MATLAB, large vectors and arrays must be allocated before use, or performance can suffer severely.

Usually, we can't count the floating point operations to tell whether our program is running at the computer's "speed limit". 1 However, we often have a formula for the amount of work as a function of problem size (in this case, the matrix dimensions M and N).

We can plot program speeds versus size, looking for discrepancies.

We can also compare two algorithms for the same problem, although it's best to use a range of input to get a more balance comparison.

SPACE: Remarks

I've been told that this example is artificially contrived to make the loops look worse, because of my choice of ordering for the two loops.

That is, I chose to write the **array1** function as

```
for i = 1 : m
  for j = 1 : n
    a(i,j) = sin ( i * pi / m ) * exp ( j * pi / n );
  end
end
```

when I could have interchanged the i and j loops. My critic suggested that the code would run faster that way. Is this true? If so, how much of a difference can it make? And if so, is there an explanation?

SPACE: Remarks

A possible explanation is that the computer stores a 2D array as a sequence of rows or columns; it is typically significantly faster to work with such an array by requesting the elements in the order in which they were stored.

MATLAB follows FORTRAN in storing a 2D array by columns. That is, the element immediately following A(1,1) in memory is A(2,1). Therefore, if you have the choice, a double loop accessing all array elements as **a(i,j)** should be written

```
for j = 1 : n
  for i = 1 : m
    a(i,j) = sin ( i * pi / m ) * exp ( j * pi / n );
  end
end
```

But does this matter? A lot?

SPACE: Remarks

I hope that by this time you realize that it can matter, and perhaps a lot, and that you also have an idea how to test this, by looking at how a fairly large array is handled.

I wrote a program which computes the array elements in both IJ and JI order (and **clears** the array in between!). I ran it without and with preallocation of the array space. It's clear that allocation is the biggest issue, but that the IJ ordering can occasionally cause a big slowdown itself.

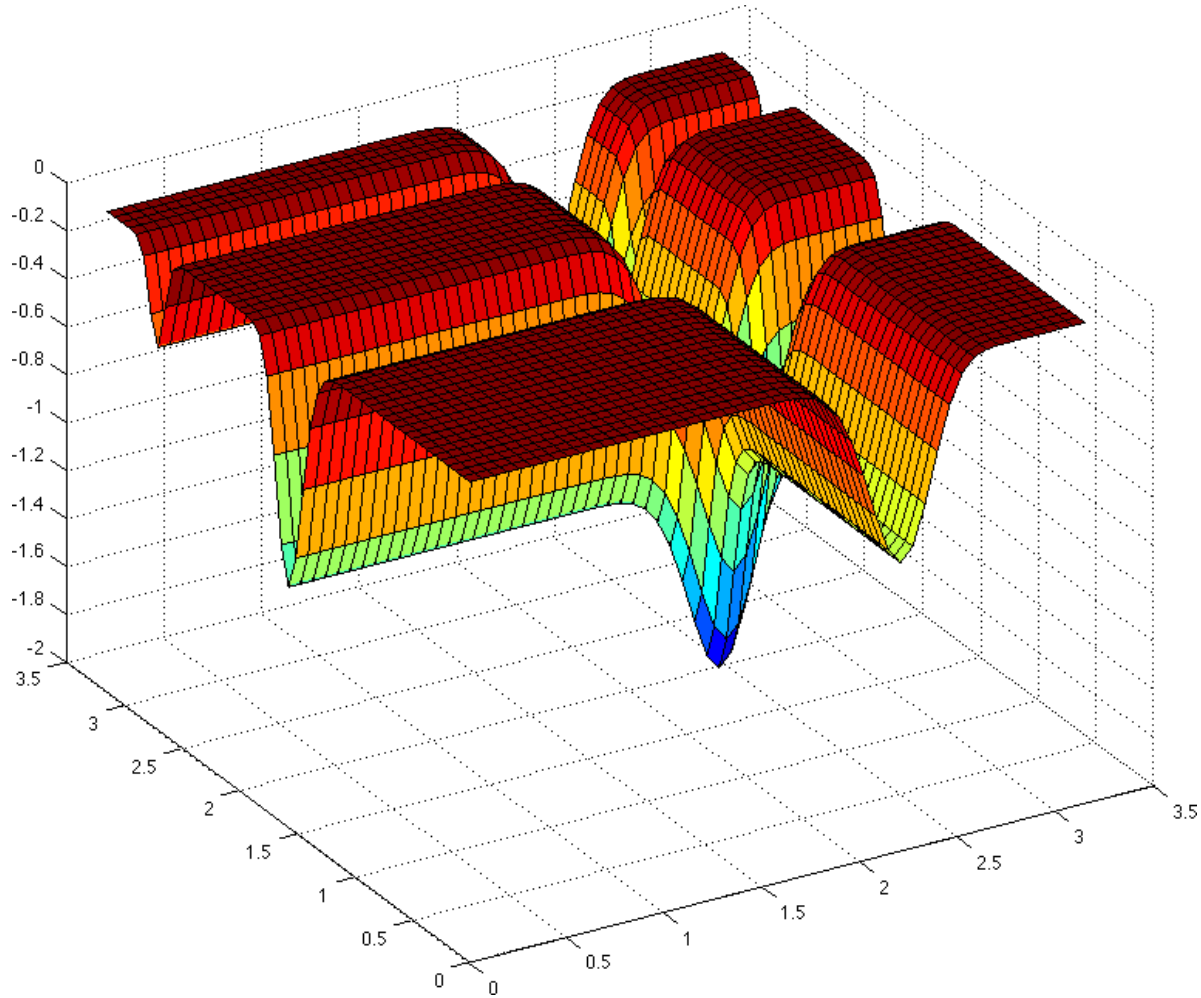
M	N	IJ	JI	IJ	JI
		no allocate		allocate	
10	100,000	3.4E4	4.4E6	7.6E6	8.1E6
1000	1,000	4.2E6	4.2E6	8.0E6	7.4E6
100,000	10	2.0E6	5.2E6	8.1E6	8.1E6

Maximum MATLAB

1. TIC/TOC
2. What's the Speed Limit?
3. Making Space
4. **Using a Grid**
5. Sparse Matrices

GRID: A Grid is a Vector, Matrix, or Array of Numbers

We often carry out computations on a rectangular grid



...and this is a natural chance to use vector operations!

GRID: Grids Are Used as Discrete Samples of Space

Because computation involves discretization, and because we have so many geometric calculations, it is often the case that we are dealing with repeated calculations at the points of a regularly spaced grid in 1, 2, 3 or more dimensions, including:

- analyzing the pixels in an image;
- approximating an integral using a product rule;
- discretizing a partial differential equation.

The commands that MATLAB has for vectors, matrices and higher-order arrays can be used to speed up such calculations.

Let's seek the minimum over a 1001x1001 grid on $[0, \pi]x[0, \pi]$ of

$$f(x, y) = -\sin(x)(\sin(\frac{x^2}{\pi}))^{20} - \sin(y)(\sin(\frac{2y^2}{\pi}))^{20}$$

(This is the function seen on the previous slide.)

GRID: Nested FOR Loops Can Generate a Grid

A natural way to code this problem would be:

```
function [ fxy_min, t ] = min1 ( n )

  tic;
  fxy_min = Inf;
  for j = 1 : n
    for i = 1 : n
      x = pi * ( i - 1 ) / ( n - 1 );
      y = pi * ( j - 1 ) / ( n - 1 );
      fxy = f1 ( x, y );
      fxy_min = min ( fxy_min, fxy );
    end
  end
  t = toc;

  return
end
```

GRID: The Function File

The function file **f1.m** looks like this:

```
function value = f1 ( x, y )

value = - sin ( x ) * ( sin (     x^2 / pi ) )^( 20 ) ...
        - sin ( y ) * ( sin ( 2 * y^2 / pi ) )^( 20 );

return
end
```

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/f1.m

GRID: MATLAB Functions Replace FOR Loops

MATLAB offers the following tools that can be used to vectorize this calculation:

- **x = linspace(a,b,n)** returns a row vector of **n** values from **a** to **b**;
- **[X,Y] = meshgrid(x,y)** returns arrays **X** and **Y** for a product grid from **x** and **y**;
- **v = min (F)** returns the minimum of each column of the array **F**.

To really speed up the calculation, we also want to call the function **f1** just one time, rather than a million times (otherwise, we pay one million times the overhead cost of one function call.) To do that, we need to rewrite the function so that it can accept a vector or array of arguments.

GRID: Vectorized Version of Function File

The revised function file **f2.m** looks like this. We have enabled this function to accept vector arguments by using the element-wise operations

```
function value = f2 ( x, y )

value = ...
  - sin ( x ) .* ( sin (     x.^2 / pi ) ).^( 20 ) ...
  - sin ( y ) .* ( sin ( 2 * y.^2 / pi ) ).^( 20 );

  return
end
```

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/f2.m

GRID: The FOR Loops Are Gone

Now we are ready to write our vectorized calculation:

```
function [ fxy_min, t ] = min2 ( n )

  tic;

  x = linspace ( 0.0, pi, n );
  y = linspace ( 0.0, pi, n );
  [ X, Y ] = meshgrid ( x, y );
  F = f2 ( X, Y );
  fxy_min = min ( min ( F ) );

  t = toc;

  return
end
```

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/min2.m

GRID: The Vectorized Code Runs Faster

We can compare our two programs, min1.m and min2.m, for a range of values of **n**, the number of points on one side of the grid:

N	min val	MIN1	MIN2
10	-0.9631	0.0007	0.0014
100	-1.7884	0.0707	0.0015
1000	-1.8012	6.7651	0.0904
10000	-1.8013	668.5089	294.6438

The **min1** timings grow as we expect, by the same factor of 100 that measures the increase in problem size.

For N=1000, we see that our new approach is 50 times faster. It's only when we look at the next line that we see a catastrophe. The **min2** is blowing up.

We're asking for an awful lot of memory, 100,000,000 real numbers at one time. Our computer doesn't have that much fast memory, so it uses slower memory, causing the performance hit.

GRID: The Vectorized Code Runs Faster

Vectorization doesn't require that we do the whole problem in one shot, but that we do the problem in sizable chunks. We can modify our program to do the 100,000,000 calculations in groups of 1,000,000, a value which the computer can handle.

Now compare the performances on the last line!

N	min val	MIN1	MIN2	MIN3
10	-0.9631	0.0007	0.0014	(0.0014)
100	-1.7884	0.0707	0.0015	(0.0015)
1000	-1.8012	6.7651	0.0904	(0.0904)
10000	-1.8013	668.5089	294.6438	8.4357

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/min3.m

GRID: Remarks

This example involved a lot of iterations of a loop, and calls to a function, both of which incur MATLAB overhead.

When we're talking thousands or millions of iterations, this overhead can dominate the calculation.

It's time to look at MATLAB vector operations, and if necessary, to rewrite your own functions in vector form, so that a loop is replaced by a vector operation, and millions of calls to a user function become just one.

If you run out of memory, you can back off and try to carry out your array operations using smaller chunks.

Maximum MATLAB

1. TIC/TOC
2. What's the Speed Limit?
3. Making Space
4. Using a Grid
5. **Sparse Matrices**

SPARSE: There's No Need to Store Zeros

What if you set aside a huge amount of space

...and almost nobody came to use it?

SPARSE: Store and Use Sparse Data Efficiently

Mathematics teaches us to ignore details. If A is a matrix, and x is a vector, then the operations of matrix-vector multiplication $y = A * x$ or of solving the linear system $Ax = b$ are important to us, but the details of how the matrix is stored seem trivial.

A sparse matrix is one in which there are a lot of zeros. If you know you're dealing with a sparse matrix, MATLAB makes it pretty easy to set up a sparse array which looks and works the same way as a regular array, but which requires much less storage, and which can be operated on much more efficiently.

So there are two issues here:

- **memory**: you only store the nonzero data;
- **speed**: you only operate on nonzero data.

SPARSE: A Sparse Array Stores Indices and Values

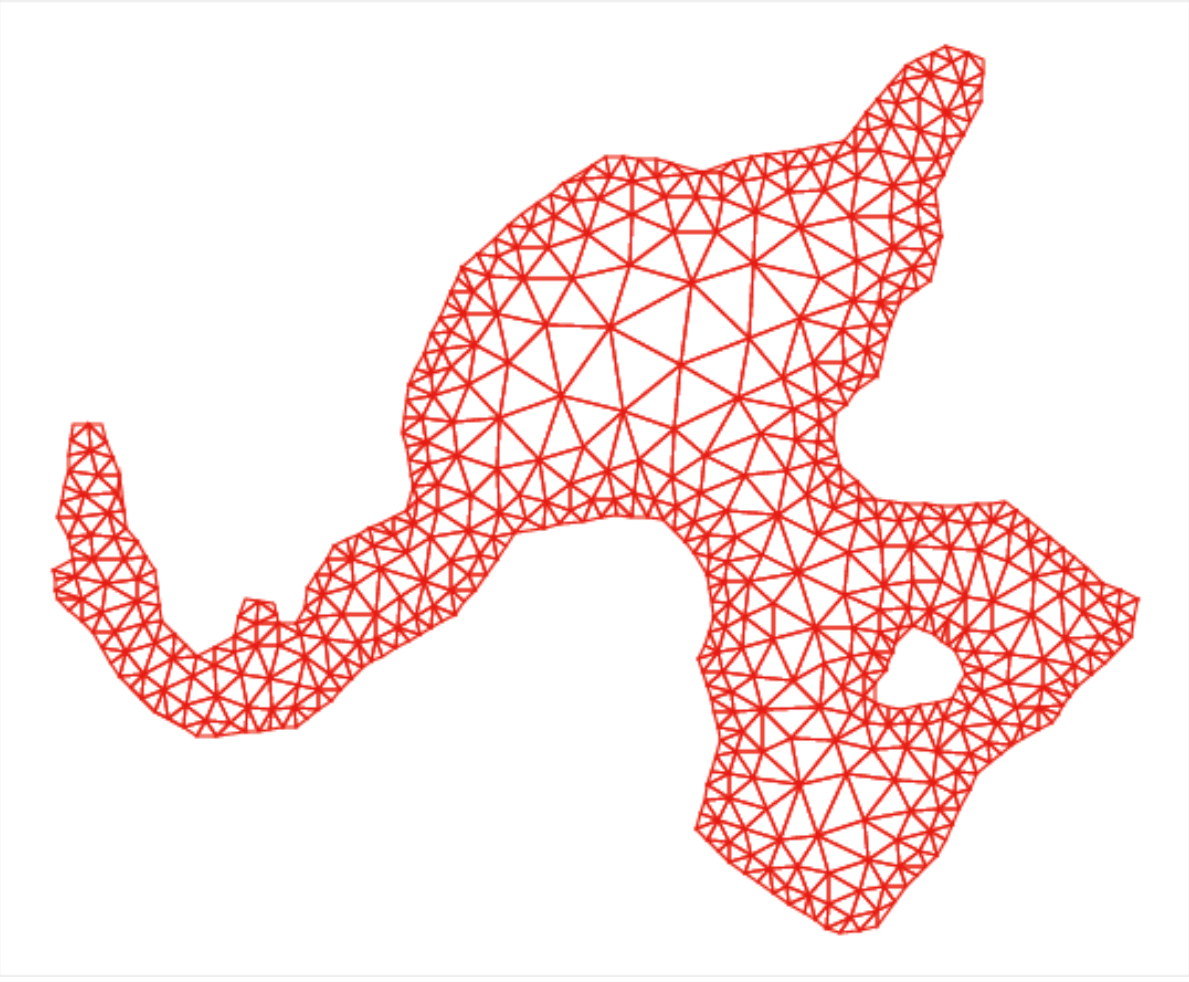
If MATLAB knows an MxN matrix is sparse, then for each nonzero entry it stores the value, and the indices I and J. This means a huge saving in storage, at the cost of some bookkeeping.

Although the matrix is not stored as a traditional array, MATLAB can quickly retrieve the information it needs for multiplication, system solving, or any other linear algebra operation.

And knowing which entries are zero means MATLAB skips many unnecessary steps. (In matrix multiplication, it doesn't need to bother multiplying by zero entries. In Gauss elimination, it doesn't need to zero out entries that are already zero.)

SPARSE: A Finite Element Mesh

Here is a relatively crude finite element mesh of 621 nodes and 974 triangular elements representing a lake with an island.



SPARSE: The Finite Element Matrix is Sparse

We are modeling pollutant diffusion in the water of the lake.

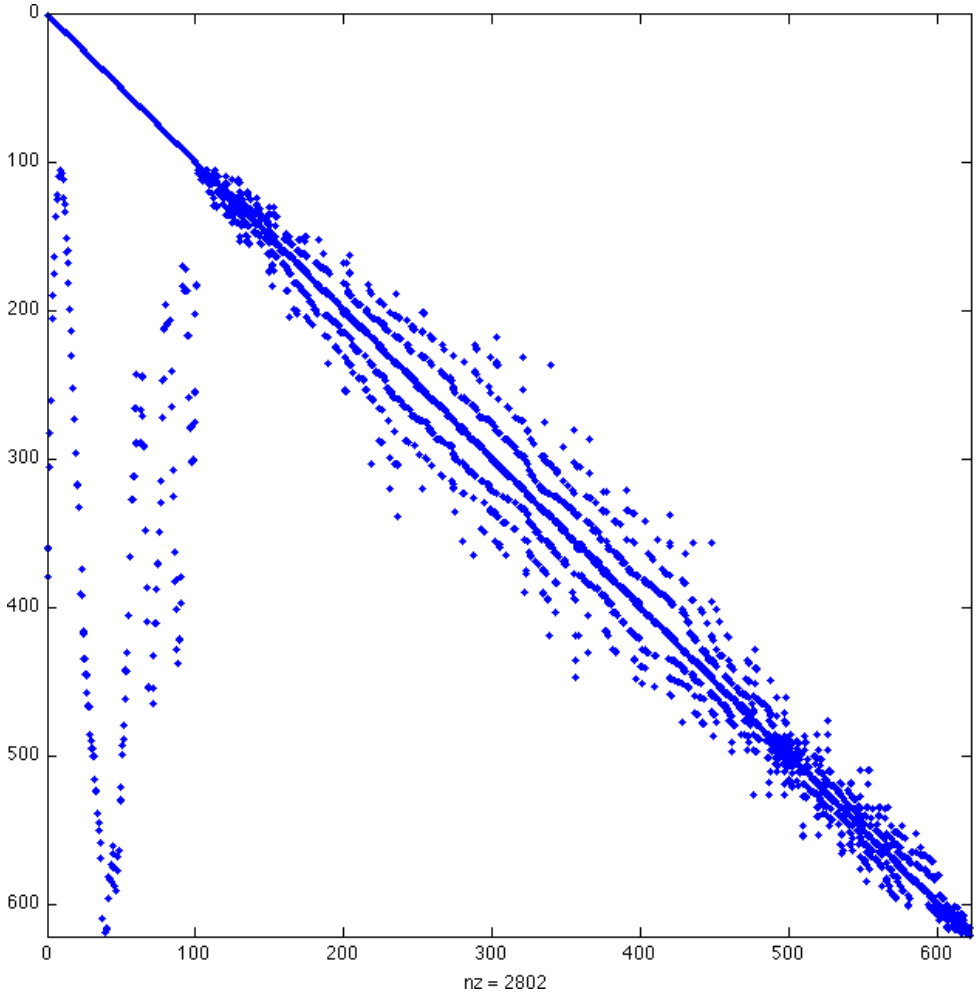
To solve the Poisson diffusion equation $-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y)$, we associate an unknown with each node, and assemble a matrix whose nonzero entries occur when two nodes are immediate neighbors in the mesh.

It is obvious from the picture that most nodes are not neighbors, and so most of the matrix will be zero. This is a common fact about finite element and finite difference methods.

It is not unusual to want to solve a problem with 1,000,000 nodes. Using full storage for the finite element matrix would require a trillion entries. Needless to say...

SPARSE: A Sparse Matrix Is Mostly Empty

Here is the sparsity pattern for our small finite element matrix, with 621 rows and columns, displayed using MATLAB's **spy()** command:



SPARSE: Use MATLAB's sparse() Command

To define a sparse matrix, you call **A = sparse()**. In the simplest case, you simply set aside enough storage, by passing the dimensions, and an (over)estimate of the number of nonzeros.

If **A** is a 100x200 matrix, with "around" 400 nonzeros, try:

```
A = sparse ( [], [], [], 100, 200, 450 );
```

I've asked for 450 entries to have room for error or growth.

The first three arguments specify the row, column and value of the nonzero elements, if you have them ready (I don't).

Once you declare the matrix to be sparse, you can put the entries in one at a time, using ordinary notation like

```
A(i,j) = v;
```

and use any MATLAB notation allowed for ordinary matrices.

SPARSE: Space and Time Comparison

For our finite element example, the matrix is 621x621.

Full storage of this matrix would require 385,641 entries for full storage. Since there are just 2,802 nonzero entries, sparse storage is much cheaper. We actually store three items per nonzero, but the cost is still just 8,406 items.

Moreover, as N increases, the full storage requirement goes up quadratically, the sparse storage linearly.

When we solve the finite element system, simply using MATLAB's "backslash" operator, the sparse system is solved 50 times faster than the full system, even though the data is identical.

SPARSE: Tridiagonal Matrices are Sparse

A tridiagonal matrix is a sparse matrix with very regular behavior. People have developed storage and solution schemes for this special case.

The discretized 1D Poisson operator becomes a tridiagonal [-1,2,-1] matrix.

The LINPACK routine **sgtsl()** factors and solves such a linear system, storing only the three nonzero diagonals. The only advantage remaining to **sparse()** might be the time required.

Let's compare **sgtsl()** and **sparse()** for a sequence of problems. Since we're only storing diagonals, we can consider NxN matrices in which N gets up to 1,000,000.

---

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/sgtsl.m

SPARSE: Comparison for Tridiagonal Matrices

N	SGTSL seconds	SPARSE seconds	FULL seconds
1,000	0.0003	0.00004	0.035
10,000	0.0028	0.0003	18.8429
100,000	0.0319	0.0037	*(too big to store!)*
1,000,000	0.2787	0.0364	*(too big to store!)*

The **sparse()** code still wins, but now only by a factor of 10, and it's possible that we could cut down the difference further.

The timings for **sgtsl()** and **sparse()** grow linearly with **N**, because the correct algorithm is used. Full Gaussian quadrature grows like N^3, so if space didn't kill the full version, time would!

http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/sgtsl_vs_sparse.m

SPARSE: Remarks

If you're dealing with large matrices or tables or any kind of array, think about whether you really need to set aside an entry for every location or not. If you can use sparse storage, you will be able to work with arrays much larger than your limited computer memory would allow.

For sparse matrices, an estimate of the number of nonzero elements is important so that MATLAB can allocate the necessary space just once.

When I do my finite element calculations, I essentially set the matrix up twice; the first time I don't store anything, but just count how many matrix elements I would create. I call sparse() to set up that space, and then I can define and store the values.

SPARSE: Conclusion

"Thank you Mary, you have entertained us quite enough."
(Pride and Prejudice)

SPARSE: Conclusion

As problem size increase, the storage and work can grow nonlinearly.

MATLAB behaves very differently depending on whether you are doing a small or big problem.

Traditional one-item-at-a-time data processing with **for** loops can be very expensive, and you should consider using vector notation where possible.

You must be very careful not to rely on MATLAB to allocate your arrays, especially if they are defined one element at a time.

If you are working with arrays, you should be aware of the **sparse()** option, which can enable you to solve enormous problems quickly.

SPARSE: More Information

A copy of this talk is available at:

`http://people.sc.fsu.edu/~jburkardt/presentations/...`
`auburn_2012.pdf`

The MATLAB codes and data are also available:

`http://people.sc.fsu.edu/~jburkardt/latex/auburn_2012/...`
`auburn_2012.html`