14:440:127– Introduction to Computers for Engineers

Notes for Lecture 10

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 What is Efficiency?

When you're programming, you ideally want short, elegant programs that run quickly, use very little space in the computer's memory, and always give the correct answer. In practice, particularly as you're learning to program, you'll sometimes write code that runs slowly. There are two main ways to make sure your code runs quickly:

1. Think deeply about the process you're using the solve a problem and consider whether the number of 'steps' in your process can be reduced.
2. Use small tricks in the programming language you've chosen that optimizes how each step is implemented in memory and on your physical computer.

We'll first discuss the second of these methods since it's much easier, albeit much less effective. The first of these methods is a large part of 'Computer Science' as a discipline.

2 Optimizing Matlab Code

In Matlab, you'll often hear that it's bad to use loops when you could have instead used vector or matrix functions. This is because Matlab's built-in functions and arithmetic operations are optimized to use data stored as vectors or matrices in memory. Thus, whenever possible, use vectors, matrices, and their associated functions. Many of the examples I've given you so far in this course could be written more efficiently using built-in functions and by 'vectorizing' (replacing loops with vector operations). I've given you many examples using loops for two reasons. First of all, they're often some of the simplest loops to write, and thus they serve a pedagogical purpose. As you start writing more complicated programs, you'll often need to write loops, but these loops will be implementing much more complex algorithms. Second, most computer programming languages use loops much more extensively than Matlab, so being good at loops is a skill-set that you can port to other languages you learn. As with natural (human) languages, learning more programming languages becomes easier after your first one or two!

2.1 Measuring Efficiency- Tic and Toc

Matlab has two convenient commands that let you time how long an operation takes to compute. To start (and reset) a timer, use the command *tic;* . To stop the timer and display the amount of time elapsed, use *toc;* . If you've brainstormed a few different methods for solving a problem and want to have an idea of which method runs most quickly in Matlab, use *tic* and *toc*!

2.2 Loops, Pre-allocated Memory Loops, Vector Operations

As an example of the huge differences in efficiency between methods in Matlab, let's compare three different methods of performing the same operation in Matlab. Particularly, let's create a 10,000 element vector filled with the number *5*.

First, we can use the totally naive method and create this vector one element at a time:

```
% Allocate memory one element at a time
clear;clc
tic
for z = 1:10000
    x(z) = 5;
end
toc
```

This method takes **0.181933 seconds** on my desktop.

Now, let's pre-allocate memory for the vector we're creating. By this, we mean that we want to create a matrix full of zeros that's the eventual size we want. That way, Matlab will set aside space in memory for the entire vector, rather than having to mess around with allocating memory each time. If you're interested in this subject, I'd recommend you take more advanced classes in Computer Engineering or Computer Science such as Architecture, Operating Systems, and Compilers.

```
% Allocate memory in advance
clear;
tic
x = zeros(1,10000);
for z = 1:10000
    x(z) = 5;
end
toc
```

This method takes **0.000111 seconds** on my desktop. In other words, it seems that the process of changing the size of a vector creates lots of overhead for Matlab. Therefore, creating data structures that are the right size (or larger) than needed makes your program run more quickly.

Now, let's use the optimized, built-in functions (and arithmetic):

```
% Use built-in functions
clear;
tic
x = 5*ones(1,10000);
toc
```

This method takes **0.000059 seconds** on my desktop, or approximately half the amount of time of using a loop!

2.3 Sparse Matrices

Space efficiency is also a concern when you're working with computers. Sometimes, you'll have a matrix filled mostly with 0's, but with a few nonzero elements. In general, these matrices would take up a lot of memory (storing thousands of zeros takes up the same amount of memory as storing thousands of different numbers. However, you can instead use a sparse matrix. Rather than storing each element of that sort of matrix, it only stores the nonzero values and their locations. You can use the *sparse()* command to convert a matrix to its sparse equivalent. Note that you should only do this when the matrix is mostly zero. Note in the example below how much less space a sparse matrix requires:

```
>> a=zeros(100);
>> a(15,26)=15;
>> b=sparse(a)
b =
  (15,26)      15

>> whos
  Name      Size             Bytes  Class
  a       100x100            80000  double array
  b       100x100              416  double array (sparse)
```

However, if the matrix has mostly unique numbers in it, you should NOT convert to a sparse matrix. Because a sparse matrix stores the value AND location of each element in the matrix, whereas a normal matrix only stores the values in sequential order, a matrix full of random numbers will require MORE space to be stored as a sparse matrix:

```
>> a = rand(100);
>> b = sparse(a);
>> whos
  Name        Size              Bytes  Class     Attributes
  a         100x100             80000  double
  b         100x100            120404  double    sparse
```

3 Case Study 1: The Efficiency of Sudoku Solving

In Fall '09, one of the projects was to write a Matlab program that could solve Sudoku puzzles. Through this example, we'll look at two entirely different ways of solving a problem:

3.1 Brute Force

A brute force algorithm would have tried all of the possibilities for each spot in the Sudoku Puzzle, as below. (Actually, here, I'm randomly trying possibilities, but it's a similar idea):

```
function p = sudokurand(p)
z = find(p==0);
doesntwork = 1;
while(doesntwork)
   doesntwork = 0;
   p(z) = ceil(9*rand(1,length(z)));
   for g = 1:9
       if(length(unique(p(g,:)))~=9 | length(unique(p(:,g)))~=9 |
                       length(unique(reshape(p((ceil(g/3)*3-2):(ceil(g/3)*3),
                       ((rem(g,3)+1)*3-2):((rem(g,3)+1)*3)),1,9)))~=9)
           doesntwork = 1;
           break
       end
   end
end
```

Note that this is not a good way to code a solution to this problem. I started running this on an easy puzzle, went and had dinner at Cinco de Mayo, came back, wrote these lecture notes... and it still hasn't solved an easy puzzle. Why? Let's say that there are 45 blank spaces in the puzzle. Thus, there are 9^{45} possible solutions to the puzzle. On average, you'd have to try $9^{45}/2$ possibilities before finding the right one. Yikes!

A better solution is possible through a method that could be described as a depth first search with backtracking, which means it starts filling in numbers from the beginning and goes as far as it can until things fall apart. When things fall apart (no numbers are possible at one of the blank spaces further down the line based on what we've guessed for the previous numbers), one of the earlier guesses must be wrong, so we go backwards and change the earlier guesses.

The idea of our function is that we'll find all of the places where the puzzle is blank (0) and only consider those. We'll keep track of which position, *pos*, we're on *among those places where there are zeros.* We'll loop until all of the zeros are filled in:

- Whatever number we're currently at, add one. If we're seeing a number for the first time, it'll be zero, so we'll try 1 first. Otherwise, if we return to a number, whatever was there previously must not have worked. We don't want to start from 1 again since we already discovered those numbers don't work. We pick up where we left off.
- If, by adding one, we're past 9, the previous positions must be wrong since 1-9 all didn't work here. Therefore, move backwards.
- Otherwise, if whatever number we just put in this spot works (yes, that's a long test), move forwards.

```
function p = sudokuiterativeshort(p)
[zr zc] = find(p==0);
pos = 1;
while(pos<=length(zr))
    p(zr(pos),zc(pos)) = p(zr(pos),zc(pos))+1;
    if(p(zr(pos),zc(pos))>9)
        p(zr(pos),zc(pos)) = 0;
        pos = pos - 1;
    elseif(sum(p(zr(pos),:)==p(zr(pos),zc(pos)))==1 & sum(p(:,zc(pos))==p(zr(pos),zc(pos)))==1 &
                sum(sum(p((3*(ceil(zr(pos)/3))-2):(3*(ceil(zr(pos)/3))),
                (3*(ceil(zc(pos)/3))-2):(3*(ceil(zc(pos)/3))))==p(zr(pos),zc(pos))))==1)
        pos = pos + 1;
    end
end
```

3.2 Recursive Sudoku

We'll learn about recursion later in this course, but I wanted to give the example of recursive Sudoku along with these examples. Our recursive idea here is that we'll try to insert a number that works. Once we find a number that works, we'll recursively call our function for this new puzzle. Our base case will be having no numbers to return, which means we're done, and we'll send back the puzzle. Otherwise, if we get stuck, we send back the empty matrix, which is our signal that we ran into a dead end.

```
function out = sudokurecursive(x)
[zr zc] = find(x==0);
if(length(zr)==0)
    out = x;
    return;
end
out = [];
for guess = 1:9
    x(zr(1),zc(1)) = guess;
    currentgroup = x((3*(ceil(zr(1)/3))-2):(3*(ceil(zr(1)/3))),
                          (3*(ceil(zc(1)/3))-2):(3*(ceil(zc(1)/3))));
    if(sum(x(zr(1),:)==x(zr(1),zc(1)))~=1 | sum(x(:,zc(1))==x(zr(1),zc(1)))~=1 |
                           sum(currentgroup(:)==x(zr(1),zc(1)))~=1)
        continue
    end
    out = sudokurecursive(x);
    if(~isempty(out))
        return
    end
end
```

4 Case Study 2: Primality Testing

Now, let's look at a number of ways of determining whether or not a number is prime and how radically our attempts differ in the amount of time they take.

4.1 Naive Primality Testing

In the most native attempt at determining whether a number X is prime, we can try dividing by all of the numbers from 2 to X-1. (Alternatively, we could stop at X/2, but that's only a slight improvement):

```
function z = prime1(x)
z = 1;
for j = 2:(x-1)
    if(rem(x,j)==0)
        z = 0;
        return
    end
end
```

4.2 Precomputing Sieve of Eratosthenes

Instead, we could try precomputing which numbers are prime using the Sieve of Eratosthenes, which you likely learned in elementary school. We'll store the vector of 1's and 0's and use it to just 'look up' whether a number is prime or not:

```
tic
z = ones(1,40000000);
j = 2;
while(j<(40000000/2))
    z((j*2):j:end) = 0;
    j = j+1;
    while(z(j)==0) % find next prime
        j = j+1;
    end
end
save primetable z; % save z to a file called primetable
toc
-----------------
function out = prime2(x)
load primetable;
out = z(x);
```

4.3 Precomputing With Sanity Checks

Of course, before taking the time to load our giant table of prime numbers, we could check whether any obvious properties of the number make it obvious that it's composite. For instance, if the final digit is an even number of 5, or if the sum of the digits is divisible by 3, we know that the number is composite and thus shouldn't bother loading the table:

```
function out = prime3(x)
z = num2str(x);
if(z(end)=='0' | z(end)=='2' | z(end)=='4' | z(end)=='5' | z(end)=='6' | z(end)=='8') % x is divisible by 2
    out = 0;
    return % if it's divisible by 2 or 5, the function is done
end
if(rem(sum(double(num2str(x))-48),3)==0) % if sum of digits / by 3
    out = 0;
    return % if it's divisible by 3, the function is done
end
load primetable; % we're only here if it's not divisible by 2 or 3
out = z(x);
```

4.4 Precomputing But Only Storing Primes

Of course, we could also notice that most numbers are composite. Thus, if we only store a list of the numbers that are prime, we'll have much less information to store:

```
tic
load primetable;
x = zeros(1,5000000); % make the assumption that we'll have fewer than 5,000,000 primes
counter = 1;
for j = 1:length(z)
    if(z(j)==1)
        x(counter) = j;
        counter = counter+1;
    end
end
x = x(1:(counter-1));
save primetable2 x; % save x to a file called primetable2
toc
-----------------
function out = prime4(y)
load primetable2;
out = sum(x==y);
```

4.5 Randomized Primality Testing

In the final and perhaps most initially bizarre (yet most useful) example, we can use a result from (theoretical) mathematics to test the primality of a number. According to Fermat's Little Theorem, $a^{p-1} \equiv 1(modp)$ for all prime numbers p and $1 \leq a < p$.

From this result, we know that if we take some number n, pick a number for a, calculate $a^{n-1}(modn)$ and get anything except 1 as our answer, n cannot possibly be prime since it would be violating Fermat's Little Theorem. Conversely, if we follow those same steps and get 1 as our answer, then n may be prime (or it may not be). In the case that $a^{n-1}(modn)$ is 1 yet n is composite, the number a is referred to as a 'false witness' to n's primality. Of course, if we have a probability ϵ that we get the wrong answer... but now repeat this process for k different randomly chosen numbers a, we have a probability ϵ^k that we wrongly claim that a composite n is prime and thus $1 - \epsilon^k$ probability that we give the correct answer. If ϵ is small, then we're VERY RARELY wrong. This method is called a *randomized algorithm* and it's important to remember that in principle, you will sometimes get the wrong answer... but with very low probability.

The Matlab implementation below runs into integer overflow errors (NaN) for larger values of n. There are workarounds to this, but it would obscure the point I'm making:

```
function out = prime5(n)
for k = 1:20
    a = ceil((n-1)*rand(1));
    if(mod(a^(n-1),n)~=1)
        out = 0;
        return
    end
end
out = 1;
```

4.6 Performance Comparison

So how does each algorithm compare? First, we test whether the number 11 is prime as a sanity check. We get the correct answer from each algorithm in the following times:

```
isprime 0.000193 seconds.
prime1  0.000125 seconds.
prime2  0.456999 seconds.
prime3  0.444343 seconds.
prime4  0.138963 seconds.
prime5  0.000121 seconds.
```

Since they needed to load lots of data, all of the precomputed functions performed poorly here. However, the naive method and the randomized method both performed better than *isprime* (likely since I left out error checking procedures). Cool.

Now, what if we try a reasonably large composite number (12345645)?

```
isprime 0.000337 seconds.
prime1  0.000118 seconds.
prime2  0.447373 seconds.
prime3  0.000297 seconds.
prime4  0.153248 seconds.
prime5  NaN
```

Again, the precomputed functions performed poorly, with the exception of the precomputed function with a sanity check (which saw 5 as the final digit and quickly realized the number was composite). The naive method of course found a factor very quickly.

However, now we test the primality of a large number that's actually prime (12345647) and discover that the naive method is BY FAR the worst in this scenario:

```
isprime 0.000383 seconds.
prime1  1.480810 seconds.
prime2  0.546471 seconds.
prime3  0.444909 seconds.
prime4  0.366813 seconds.
prime5  NaN
```

Since a ton of steps were taken in dividing 12345647 by tons of different numbers before finding that none of them evenly divided 12345647, that method performed quite poorly.

In the end, which method is best? Probably a hybrid that performs some obvious sanity checks (looking at the last digit) first before using a randomized algorithm or perhaps a precomputed table, maybe containing only large prime numbers (and using the naive method for smaller numbers). This is where the engineer in you combines theory and practice!

5 Alternative Data Structures

Structures, or "structs," are a very important data type when working with databases, or even with GUIs (handles are a struct). A structure allows you to associate numerous "fields" with one main variable. You access these fields by stating the variable name, followed by a period, followed by the field name:

```
person.iq = 40;
person.name = 'Soulja Boy';
person.home = 'Bad Radio';
person.pet = 'Fido Tell ''Em';
```

Now, the variable *person* will contain "iq", "name", "home" and "pet" fields. You could type something like *person.iq*2*, which would give the answer 80. Why is it useful to have all of these fields? Well, if you're creating a database, you could create the following vector of structures, and then use a loop to print them out:

```
person(1).iq = 40;
person(1).name = 'Soulja Boy';
person(1).home = 'Bad Radio';
person(1).pet = 'Fido Tell ''Em';
person(2).iq = 120;
person(2).name = 'Kurt Cobain';
person(2).home = 'WA';
person(2).pet = 'Polly';
person(3).iq = 130;
person(3).name = 'Miles Davis';
person(3).home = 'NY';
person(3).pet = 'Cat';

for p = 1:3
   fprintf('%s lives in %s with a pet %s and a %.0f iq \n',
            person(p).name,person(p).home,person(p).pet,person(p).iq)
end
```

6 Binary Numbers

To introduce the topic of data types, we first need to introduce binary numbers, or numbers base 2. You're used to seeing numbers written "base 10." In this system, the rightmost digit is the 1's place, then to the left you get the 10's place, the 100's place, the 1000's place, etc. In each of these places, you can have the numbers 0 through 9. Where do you get 1's, 10's, 100's, 1000's? Well, $1 = 10^0, 10 = 10^1, 100 = 10^2, 1000 = 10^3$, etc.

In base 2, or binary, the only possible digits are 0 and 1. Rather than having the "places" be the powers of 10, they are instead the powers of 2. Thus, the rightmost place is still the 1's place ($1 = 2^0$), but the next place to the left is now the 2's place ($2 = 2^1$). Then, the next place to the left is the 4's place ($4 = 2^2$), then the 8's place ($8 = 2^3$), and so on.

You can convert between Base 2 and Base 10, as in this example:

```
What is 10110101 (base 2) in base 10?
Let's first label the places:
---  ---  ---  ---  ---  ---  ---  ---
128   64   32   16   8    4    2    1
2^7  2^6  2^5  2^4  2^3  2^2  2^1  2^0

So, our number, with the places labeled below it, is:
_1_  _0_  _1_  _1_  _0_  _1_  _0_  _1_
128   64   32   16   8    4    2    1

Now, let's write this in expanded notation:
1*128 + 0*64 + 1*32 + 1*16 + 0*8 + 1*4 + 0*2 + 1*1
1*128 + 1*32 + 1*16 + 1*4 + 1*1
128+32+16+4+1 = 181

Thus, 10110101 (base 2) is 181 (base 10)
```

Now, let's see how to go in the other direction:

```
What is 87 (base 10) in base 2?
Let's first label the places in base 2 again:
---  ---  ---  ---  ---  ---  ---  ---
128   64   32   16   8    4    2    1
2^7  2^6  2^5  2^4  2^3  2^2  2^1  2^0

Put a 1 in the FIRST PLACE TO THE LEFT
that is smaller than 87.

So our number, with the places labeled below it, is:
_0_  _1_  ---  ---  ---  ---  ---  ---
128   64   32   16   8    4    2    1

We've now accounted for 64 of the 87, so we have
23 (87-64) left.  So, put 0's in the places until you
find one less than 23, at which point put a 1.

_0_  _1_  _0_  _1_  ---  ---  ---  ---
128   64   32   16   8    4    2    1

Now, we have 7 (23-16) left.  Note that the last three
places add up to 7, so put 1's there and 0's elsewhere.

_0_  _1_  _0_  _1_  _0_  _1_  _1_  _1_
128   64   32   16   8    4    2    1

Thus, 87 (base 10) is 1010111 (base 2).
```

With computers, a "bit" is a single binary digit (a 0 or a 1). A "byte" is 8 bits, or an 8 digit number. A kilobyte can be either 1,000 bytes, or 1,024 bytes, depending on whether you want to be shady (1,000) or honest (1,024). Note that all computer manufacturers take the shady route. A megabyte is either 1,000 kilobytes, or 1,024 kilobytes. A gigabyte is 1,000 megabytes, or 1,024 megabytes. A terabyte is 1,000 gigabytes, or 1,024 gigabytes. After that, you get to petabytes, but in the next few years, none of you can download enough bootleg movies and music from DC++ to need petabytes of hard drive space. Or so I hope.

7 Data Types and Computational Error

What we mean by datatypes in Matlab are the different ways of storing numbers and information. It used to be that how you stored a number mattered since memory in a computer was at a premium. Now, that's really only the case if you're working with HUGE datasets. However, Matlab still has many different ways of storing numbers.

7.1 Integers

Integer data types (uint8, int8, uint16, int16, uin32, int32, uint64, int64) are, respectively data types that can store unsigned and signed integers using 8 bits, 16 bits, 32 bits, and 64 bits. "Unsigned" means that we assume the number is positive; on the other hand, "signed" uses the first bit of the number to store a positive or negative sign. Thus, a signed data type stores number only about half as large as an unsigned data type. However, an unsigned integer cannot store a negative number; it stores i.e. `uint32(-5)` as 0.

With your knowledge of binary, you can calculate the largest and smallest integers that can be stored. (Note that *intmax*, the largest number, will generally be 1 smaller in absolute value than *intmin*, the smallest number. For instance, uint8 can store only as high as 127, but as low as -128. Why? Well, 'positive' 0000000 is defined as 0. You can define 'negative' 0000000 as -1, and thus get yourself one extra negative number. There's no sense in wasting that free number, right?

```
What is the largest uint8 number?

Let's first label the places in base 2 again:
---  ---  ---  ---  ---  ---  ---  ---
128   64   32   16   8    4    2    1
2^7  2^6  2^5  2^4  2^3  2^2  2^1  2^0

Put a 1 in each place.  In other words, 11111111 is the largest number.
In base 10, that is 255.  Of course, the smallest number is 0.
```

```
What is the largest int8 number?

Let's first label the places in base 2 again:
_X_  ---  ---  ---  ---  ---  ---  ---
128   64   32   16   8    4    2    1
2^7  2^6  2^5  2^4  2^3  2^2  2^1  2^0

The first place is taken up by the positive/negative sign.
Put a 1 in each other place.  In other words, 1111111 is the largest number.
In base 10, that is 127.  Recall that the smallest number is thus -128.
```

Note that in order to store a number as an integer data type, you need to explicitly "type-cast" that number. This means that you put the name of the data type as you would a function name: `int32(2^30)` or `double(12)`. You can very clearly see the limitation of how computers store numbers by doing `uint256(255)`.

7.2 Floating Point

By default, all numbers in Matlab are stored as double-precision floating point numbers. Indeed, all decimal values must be stored as floating point numbers. The idea of floating point arithmetic is similar to scientific notation in that we want to have accurate numbers (with a large number of significant digits) that can range from extremely tiny (close to 0) to extremely large in absolute value. Thus, the numbers are stored in what can be considered a perversion of scientific notation. By default, numbers you type into Matlab are stored as "doubles," or "double-precision floating point numbers." This means that there are 64 bits (digit) that can be used to represent the number.

The first bit of the 64 is used to store the sign (0 is positive, 1 is negative). The next 11 bits are used to store an exponent. 11 bits can store numbers from 0 to 2047. Since we want negative exponents, we subtract -1023 from whatever number is stored. Thus, to store an exponent of 1, we actually store 1024. The two exceptions are if we store 0 as the exponent (which, if the rest of the floating point number is 0, is assumed to be 0) or if we store 2047 as the exponent (and 0 as the rest of the number), which is infinity. Therefore, possible exponents thus range from -1022 to 1023.

The final 52 bits are used to represent the decimal part of the number; this is called the significand, fraction, or mantissa. Of course, there's another trick here. We assume that the first digit is 1 (which is always the first signficiant digit in binary), and then we use the 52 digits to represent negative powers of 2, beginning with 1/2. Thus if the fractional part is 1101, we have 1 (implicit) + 1/2 + 1/4 + 1/16 (there's a 0 in the 1/8 place). Therefore, the fractional part ranges from 1 to just under 2. Note that if the fractional part were less than one, we could just move over one exponent. If the fractional part were 2 or greater, we could move the exponent in the other direction. In the end, we take the fractional part, and get a number between 1 and 2; we then multiply that by 2 to the power of (the exponential part). Fun, right?

An important point is that any operations involving really small numbers and really large numbers will get messed up. In the following examples, the first subtraction we do involves two large numbers, and we get 0. Thus, when we add 5, we're dealing with two small numbers. This is not the case in the later examples, and we start getting WEIRD (and incorrect) results. This is due to the imprecision of our data type and is something you need to think carefully about when you are working with small and large numbers in the same calculations. (There are many methods you would learn about in a Numerical Methods class, including ideas such as Pivoting in Matrices, which makes sure you're dealing with the larger numbers first):

```
>> 2^50 - 2^50 + 5
ans =     5

>> 2^55 - 2^55 + 5
ans =     5

>> 2^60 - 2^60 + 5
ans =     5


>> 2^50 + 5 - 2^50
ans =     5

>> 2^55 + 5 - 2^55
ans =     8

>> 2^60 + 5 - 2^60
ans =     0

Say WHAT?
```

However, more subtle errors can happen. Because doubles/floats aren't stored as exact values, Matlab sometimes doesn't recognize that a particular number stored as a double is that number. For instance, let's look at the following example that actually came up as a mistake in an earlier lecture. We're going to make a vector that should contain the value 3.4, and then find where 3.4 is in that vector:

```
>> y = 3:0.1:4;
>> find(y==3.4)
  ans = 5
>> y(5)-3.4
  ans = 0

>> x=0:0.1:10;
>> find(x==3.4)
  ans = Empty matrix: 1-by-0
>> x(35)
  ans = 3.4000
>> x(35)==3.4
  ans = 0
>> x(35)-3.4
  ans = 4.4409e-016
```

Ok, now Matlab is just messing with us. When we make a vector from 3 to 4 in steps of 0.1, we can find 3.4. However, when we make a vector from 0 to 10 in steps of 0.1, we can't find 3.4 since it's 10^{-16} away from 3.4 because of the way it's being stored. Seriously, wtf?! I guarantee you that these sorts of subtle errors will drive you totally nuts when you encounter them. The reason I'm showing you how data is represented is so that you can try and find these sorts of errors and understand why they're happening.

8 Converting Data Types

We can figure out whether or not a variable X is a certain data type through the following operations, each of which returns either 1 or 0 (for true and false):

```
iscell(X)  % checks if X is a cell array
ischar(X)  % checks if X is a string or single character
isempty(X)  % checks if X is the empty vector
isfloat(X)  % checks if X is a floating point/double
isinteger(X)  % checks if X is an integer DATA TYPE
    % note that isinteger(23242) will return false
    % since numbers are stored as doubles by default
isnumeric(X)  % checks if X contains only numbers
```

When we want to specify how data is stored in Matlab, we can do what's called typecasting:

```
x = uint16(x);   % store x as an unsigned 16 bit integer
y = int16(x);  % store y as a signed 16 bit integer
z = double(z); % store z as a double (floating point)
```

We can also typcast to a string using *char*. Note that if we use *char* with numbers, it looks up that number in the ASCII chart, which follows our example:

```
char(49)
  ans = 1
char(65)
  ans = A
char(65:96)
  ans = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_‘
char(13);  % this is the newline character \n
```

http://www.jimprice.com/ascii-0-127.gif

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	A	097	a
002	☻	STX	034	"	066	B	098	b
003	♥	ETX	035	#	067	C	099	c
004	♦	EOT	036	$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	(beep)	BEL	039	'	071	G	103	g
008	◘	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	*	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	'	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♫	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	►	DLE	048	0	080	P	112	p
017	◄	DC1	049	1	081	Q	113	q
018	↕	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	¶	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↨	ETB	055	7	087	W	119	w
024	↑	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	<	092	\	124	|
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	_	127	⌂

9 Binary Number Conversion

Let's write Matlab functions to convert from base 2 to base 10 (binary to decimal). We'll assume they give us the number as just an integer or double of 1's and 0's. We'll cheat and turn the binary number into a string. That way, we have access to each digit separately since a string is just a vector of single characters. Of course, we don't want to work with characters, we want to work with numbers... so we turn each element of the character vector back into a number by subtracting 48. (Why? The 48th character in the ASCII chart is 0, the 49th is 1, and so on.) At that point, we just multiply all of the digits by their place value, which are the declining powers of 2. If you write this function, you can call it through something like *convertbase2to10(1010010101)*.

```
function base10 = convertbase2to10(base2)
coeffs = double(num2str(base2)) - 48;
base10 = sum(coeffs.*(2.^(length(coeffs)-1:-1:0)));
```

Now let's go in the other direction, converting from base 10 to base 2. Basically, we just allocate the correct amount of space in a character string of 0's. Then, we go from the largest digit to the smallest, filling in a 1 whenever *base10* is bigger than the value of that place:

```
function base2 = convertbase10to2(base10)
base2 = char(48*ones(1,floor(log2(base10))+1));
for z = 1:length(base2)
    if(base10>=2^(length(base2)-z))
        base10 = base10 - 2^(length(base2)-z);
        base2(z) = '1';
    end
end
base2 = str2num(base2);
```