Key Concepts: Economic Computation and Matlab, Part I

Brent Hickman[*]

Summer, 2008

Required Readings:

- ***Kermit Sigmon's Matlab Primer*** on the web at
  http://web.ift.uib.no/Fysisk/Teori/KURS/WRK/mat/singlemat.html

All math camp students should have already read through each page in the tutorial (except the ones on "Flops and Etime" and "Hardcopy") and tried out the examples. If you haven't finished the preliminary reading yet, please do so as soon as possible.

This short course is designed to give the student enough exposure to numerical computation and programming in MATLAB, so as to enable him/her to complete quantitative assignments that will come up in the PhD Economics core sequence (Primarily macroeconomics, dynamic programming, and econometrics). Another objective of this course is to expose the student to enough related ideas so that he/she will know where to look for additional information later on in the research phase. This is the reason for the brief introduction to AMPL in Computation Notes, Part IV.

---

[*]University of Iowa `Brent-Hickman@UIowa.Edu`

Optional/Additional Readings:

- General Computation and Algorithmics for Beginners:

    – *The Binary Revolution: The History and Development of The Computer* by Neil Barrett, WN, 2006.

    – *Computers Ltd: What They <u>Really</u> Can't Do* by David Harel, Oxford University Press, 2000 (this is the abridged version of an introductory computer science text, *Algorithmics*, also by Harel).

- Numerical Methods in Economics:

    – An excellent text for beginners is: *Applied Computational Economics and Finance* by Mario J. Miranda and Paul L. Fackler, MIT Press, 2002.

    – A more advanced text is: *Numerical Methods in Economics* by Kenneth L. Judd, MIT Press, 1998.

<div align="center">

<u>The Purpose of Computation:</u>
</div>

<div align="center">

to perform **automatable** tasks **faster**, more **efficiently** and more **accurately** than humans are capable.
</div>

<div align="center">

♦
</div>

<div align="center">

"Computers WORK; people THINK."

–Old IBM Corporation Adage
</div>

<div align="center">

♦
</div>

<div align="center">

"Man is a slow, sloppy and brilliant thinker; the machine is fast, accurate and stupid."

–William M. Kelly (American inventor and industrialist)
</div>

<div align="center">

♦
</div>

<div align="center">

"People think computers will keep them from making mistakes. They're wrong. With computers you make mistakes faster."

–Adam Osborne (inventor of the first portable computer)
</div>

<div align="center">

♦
</div>

I. Computing 101

    A. Components of a computer: **What makes it run fast?**

    Think of a computer as a laborer with a workbench...

        <u>**HARDWARE SPEED**</u>:

        1. Processor – reads and executes instructions

            ▶ Processor speed determines the number of operations performed per unit time.

            ⇒ Analogous to how skilled or how smart the laborer is

        2. Memory (RAM) – stores instructions, input data and output data

            ▶ Amount of RAM determines the quantity of data that can be manipulated at once.

            ⇒ Analogous to the size of the workbench: no matter how skilled, the laborer can't work on something unless it's right in front of him

        <u>**ALGORITHMIC SPEED**</u> **(YOUR job, as the programmer!)**:

        3. Algorithms/Programs – a sequenced set of instructions to convert an input into a desired output. Computer programs are concrete incarnations of algorithms, which are more general and abstract.

            ▶ Determines the total quantity of operations and memory required to convert inputs into outputs.

            ⇒ Analogous to the efficiency of the laborer's methods

            ▶ EXAMPLE: lookups in an ordered list of 1,000,000 items.

            Naive Search *vs.* Divide&Conquer

            1,000,000 operations *vs.* 20 in the worst case

II. HOW COMPUTERS AND HUMANS COMMUNICATE

    A. Machine Language

        1. Basic data unit: bit (short for "binary digit")

        2. Binary or base-2 representation: $9 = +0000000000000000000000000001001$

        3. Finite Precision (32-bit machine):

- ► Any integer up to $(2^{31} - 1)$ ("machine limit")

- ► Any decimal up to 16 digits ("machine precision")

4. Alphanumeric data & instructions may also be represented in binary:

- ► non-numeric characters; $+/-/\times/\div$; Input/Output instructions; Control instructions

B. Compiler/Interpreter Languages

1. A Compiler/Interpreter language is one that is formed from syntax that is more intuitive to humans than machine language. The interpreter or compiler serves as the mediator between human and machine.

2. An interpreter or compiler is a program that receives a program from one language as an input, and produces a program in another language as an output.

- ► Compiler language: converts instructions prior to execution (*e.g.*, C++)

- ► Interpreter: converts instructions <u>during run time</u> (*e.g.*, MATLAB)

- ► Although they both detect syntax errors equally well, the Compiler *vs.* Interpreter question has implications for methodological error detection and de-bugging.

- ► **Both** Compilers and Interpreters create inefficiencies/redundancies when converting instructions from a programming language to machine language. They do their job as best they can, but they don't necessarily do it optimally.

1. Early languages: Fortran, Cobol, Lisp

2. Newer languages: C/C++/C#, Perl, Java

D. High-Level Languages (don't be fooled by the fancy name!)

1. Written in another non-machine language

2. Multiple compilations before execution, exacerbating the inefficiency/redundancy problem

3. More user-friendly and easier to learn than low-level languages

4. More restrictive and less flexible than low-level languages

5. Examples of high-level languages: MATLAB, **Octave**[1], R, GAUSS, STATA, Shazam, TOMLAB, AMPL, GAMS

III. MATLAB SPECIFICS

PROS:
1. Versatility/Flexibility: includes many important features of low-level programming languages

   ▶ flow control ("looping"), subroutines

2. User-friendliness

   ▶ well-developed GUI

   ▶ built-in text editor

   ▶ "black box" features: toolboxes, extensive built-in functionality

3. Generality/portability: widely used in the economics profession, engineering and science

4. Efficiency: very fast if used properly (WITH SOME IMPORTANT CAVEATS...)

5. Adaptability: MATLAB allows for user augmentation of its built-in library of functions:

   (a) Path specification

      ▶ Store your own MATLAB functions in .m files

      ▶ In MATLAB, click File→Set Path→Add Folder, and browse to the location of the folder where your function is located. Lastly, move the new path to the bottom of the list.

      ▶ All functions in that folder may now be executed with a simple function call from any directory, just like a built-in MATLAB function.

6. EXCELLENT graphics generating capabilities in 2-D or 3-D.

7. EXCELLENT documentation and support (***THE MOST IMPORTANT LESSON FOR PART I OF THESE NOTES***!!!)

---

[1] By the way, Octave is a free, open source programming language which is designed to be MATLAB-compatible. For more information, visit http://www.gnu.org/software/octave/.

- ▶ Type `help fun` in the command line to retrieve information on the usage of function `fun`.

- ▶ Type `doc fun` to retrieve the same information in the help files, plus more.

- ▶ Search/Browse through the extensive help files.

- ▶ Online forums and file exchanges for additional support.

8. Algorithmic inefficiency detection

   - ▶ "M-Lint" utility — a built-in programming tutor

   - ▶ Profiler — analyzes code to determine where run-time bottlenecks occur and produces a detailed report

9. Seamless parallel processing for dummies with the Parallel Processing Toolbox (PPT)

   - ▶ Can use multiple processor cores on a single machine as a pool of autonomous workers to perform independent tasks simultaneously

   - ▶ The PPT automatically parallelizes your programs for you

   - ▶ Here is a recorded webinar on the MathWorks website which covers this topic in depth.

   - ▶ PPT is expensive, but very useful in certain circumstances

CONS: 1. Relatively inefficient memory management/allocation (as compared to, say R)

   - ▶ Excessive variable retention during run time

   - ▶ Type `help memory` or `doc memory` on MATLAB command line.

   - ▶ REMEDY: use the `clear` command in your programs to discard variables from memory after they are no longer needed.

2. MATLAB code is designed for use in a broad range of settings, with the tradeoff that it will not necessarily run optimally in a given setting. You can counter this when needed by designing your own streamlined code in a low-level language, tailored for low computational overhead in your specific problem.

▶ REMEDY: execute portions of your program with a `.mex file` (short for "Matlab executable") written in FORTRAN, C or C++

(a) Determine where computational bottlenecks occur with the profiler and optimize Matlab code

(b) If more speed is needed and parallelizing is not an option, write a program in say C++ to perform the segment of computation that is the bottleneck.

(c) Once your C++ subroutine is written and debugged, compile a Matlab EXecutable version of the program using the `mex` command. This creates a `.mex` function which can then be called from within a block of Matlab code, just like any built-in function. type `help mex` on the command line for more information.

3. Matlab's Optimization Toolbox does very well for linear problems; VERY POORLY for non-linear ones (*i.e.*, constrained optimization with non-linear constraints)

▶ Non-linear solvers (`fmincon`) are inefficient and unreliable for problems where the constraint set is highly non-linear

▶ Inefficient/unstable differentiation methods (finite differences)

▶ REMEDY #1: For non-linear optimization, access the most current & best non-linear solvers (KNITRO, MINOS, SNOPT, *etc.*) via TOMLAB, AMPL[2], GAMS, or C++.

▶ REMEDY #2: Starting with Matlab 2008a, the KNITRO solver can be called from within the Optimization Toolbox. You can obtain a trial version of KNITRO from Ziena Optimization, Inc.

▶ Differentiation is still more efficient within a modeling language like AMPL, due to a technique called "automatic differentiation" which takes advantage of the fact that a function and its derivative often have common components. For example, with $f(x) = e^{-x^3}$, we have $f'(x) = -3x^2 e^{-x^3}$ and $f''(x) = (-3x^2)^2 e^{-x^3} + -6x e^{-x^3}$. Rather than

_____

[2]Available in a (somewhat limited) student version for free or in unlimited form via the NEOS SERVER.

needlessly computing the terms e^{-x^3} and $-3x^2$ multiple times, it is quicker to compute them once and re-use the results multiple times.

IV. Numerical Efficiency in MATLAB

    A. MATLAB = "MATrix LABoratory"

        ▶ MATLAB was designed and named for its ability to work lightning fast with matrices.

    B. A word on use of matrices and `for` loops...

        ▶ MATLAB is like an axe with a one-sided head. You can repeatedly strike a log with either the blunt side or the sharp side of the axe head and, eventually, it will split, regardless. However, one of the two methods involves time and pain increased by several orders of magnitude, to produce the same output!

        ▶ Kernel smoothing example

        ▶ Inefficiencies/redundancies arise due to multiple compilations prior to execution: the drawback of all high-level languages.

        ▶ MATLAB source code is designed to minimize the inefficiencies/redundancies involved in **matrix operations**. <u>You should take advantage of the speed and elegance built into the language.</u>

    C. MORAL OF THE STORY:

MATRIX OPERATIONS ARE THE NAME OF THE GAME
(`for` loops are to be used SPARINGLY!)

BRENT'S OFFICIAL MATLAB RULES OF THUMB:

♦

#1: If you can build/manipulate arrays of variables using either iterative loops or matrix operations, use matrix operations instead.

♦

#2: If you think you can't build/manipulate a particular array using matrix operations, you haven't thought hard enough!!!

8

$\blacklozenge$

D. Two main advantages of favoring matrix operations over iterative loops:

   1. Computations will run faster

   2. Your code will be much more concise, understandable and <u>EASIER TO DEBUG</u>.

V. Matrix utilities in MATLAB or *"How to chop wood with the sharp side of the axe head"*

- Matrix–building utilities

  ▶ Manual input and concatenation

  ▶ `ones`

  ▶ `zeros`

  ▶ `eye`

  ▶ MATLAB's built–in indicator function

  ▶ `rand`

  ▶ the ":" operator

  ▶ `linspace`

- `find`

- Truncating a vector: the "empty–brackets" operator

- Vectorizing a matrix: the ":" operator revisited

- `reshape`

- `size`

- `length`

- `end`

- `unique`

- SORTING

  ▶ `sort`

▶ `sortrows`

- Submatrices: the ":" operator revisited

- Arithmetic matrix operations: the "." operator

- Arithmetic matrix operations: the "\" or "/" operators (see also `mldivide` or `mrdivide`)

  ▶ DO NOT use `inv` function for systems of equations.

VI. Tips on Using `while` and `for` Loops in MATLAB

1. BE MINIMAL. Remember that every operation done inside of a loop will be repeated many times. Before including anything inside of a loop, ask yourself, "is it necessary for this operation to be performed more than once?" If the answer is no, then place it outside of the loop instead.

2. USE `for` LOOPS SPARINGLY. This is especially true when constructing matrices. Before you place a `for` loop in your code, ask yourself "can I get the job done using matrix operations instead?" If the answer is no, then there's a high likelihood that you haven't thought hard enough about it. Get creative. If there's a utility that you can think of in your head to serve your purpose, chances are that someone has already coded it into MATLAB and you have only to find it in the documentation. You may also have to use some linear algebra knowledge to get the job done, but at the end of the day it will have been well worth it. Your program will run in a fraction of the time and your code will have many fewer lines, saving *a lot* of time during the debugging process.

3. USE `for` LOOPS MAINLY FOR ONE–SHOT TASKS. Don't get me wrong— `for` loops are useful for many things; but they are also inappropriate for many things. It is reasonable in some cases to use `for` loops for tasks that are only performed once, such as rearranging a data matrix in a certain way. However, you should avoid placing `for` loops inside of `while` loops or other `for` loops whenever possible. More often than not, this will lead to a slow–running, memory-intensive program, especially for beginning programmers.

VII. Final Tips on Programming in MATLAB:

1. Use informative naming conventions

- For a price elasticity of consumption good x, name it something like `elast_xp`, rather than something uninformative like `z`.

2. Include plenty of comments in your code with the `%` character. This will make it so you can understand it when you come back to it later.

3. Use cell mode in the MATLAB text editor to isolate problems or to run only a portion of your code (Cell→Enable Cell Mode). Separate cells with the `%%` character.

4. Re-use your code. Don't program the same routine over and over if you don't have to. Copy and Paste blocks of code from one program to another. Design your functions so that they can be called by future programs, not just the one on which you are currently working.

5. Pre-allocation of memory: MATLAB does not require the programmer to define a variable before assigning a value to it like many other languages (*e.g.*, C++). However, it is often more efficient to do so anyway. For example, in cases where, say the values of a matrix M are computed one-by-one within an iterative loop, many MATLAB beginners start by defining M as an empty matrix, and then tack on an additional element to M with each iteration. This is very inefficient though, because every time the dimensions of M change, MATLAB incurs a significant computational cost to re-define the matrix. If the final dimensions of the matrix are known *ex ante*, say $1,000,000 \times 1$, then a better way is to pre-allocate the memory space for M by one of the following commands: `M=zeros(1000000,1)` or `M=nan(1000000,1)`. This way, MATLAB is not forced to redefine the dimensions of the matrix multiple times; rather, it need only change the values of the existing elements of M.

6. Logical indexing versus the `find` function: The `find` function is useful for many things in MATLAB, but it is not always the best. Use of the `find` utility should be limited to situations where there is a need to compute a list of indices corresponding to elements of a matrix where a logical statement is

satisfied. However, if only the elements of the matrix are important, it may be faster to use logical indexing. For example, suppose we have a $m \times n$ matrix M with some positive and some negative entries. If it is useful to keep track of the negative entries of M, then one could use the command `index = find(M<0)`. However, if it is only necessary to perform a one-time operation, like setting all negative values to zero, then it is faster to use `M(M<0)=0` than `M(find(M<0))=0`. The former method is called logical indexing.

VIII. Analytic *vs.* Numerical Mindsets

    A. Numerical Stability: small changes to inputs should not lead to unexpectedly large changes in outputs

- Machine limit/roundoff error

- In Theory, for any singular matrix there is a non-singular matrix which is arbitrarily close in terms of the Euclidean metric. However, to a computer, around every non-singular matrix there is a ball of positive radius for which every member is singular. This is due to roundoff errors.

    ▶ DIGRESSION ON CONDITION NUMBER

- Large numbers *vs.* "`NAN`" or "`inf`". EXAMPLE: computing the binomial coefficient $\binom{n}{k}$

- Small numbers *vs.* zero. EXAMPLE: Matrix inverses and condition number

    B. *How* you compute is as important as *what* you compute

      1. EXAMPLE: Computing the sample average

        ▶ $\sum_{t=1}^{T} \frac{x_t}{T}$ or $\frac{1}{T} \sum_{t=1}^{T} x_t$?

      2. EXAMPLE: Likelihood *vs.* log-likelihood

        ▶ $L(\theta|\mathbf{x}_T) = \prod_{t=1}^{T} f(\theta|x_t)$ or $\mathcal{L}(\theta|\mathbf{x}_T) = \sum_{t=1}^{T} \ln(f(\theta|x_t))$?

      3. EXAMPLE: First-price auction differential equation

        ▶ $\beta'(v) - \frac{(v-\beta(v))f_V(v)}{F_V(v)} = 0$ or $\beta'(v)F_V(v) - (v - \beta(v)) f_V(v) = 0$?

      4. EXAMPLE: matrix inverses *vs.* Cholesky decomposition

▶ A natural way to solve the system $A\mathbf{x} = \mathbf{b}$ is to compute $A^{-1}\mathbf{b}$, but this is numerically slow and unstable.

Instead, try the following:

If A is symmetric and positive definite, then we can solve the system $A\mathbf{x} = \mathbf{b}$ by first computing the Cholesky decomposition[3] $A = LL^{\top}$, then solving $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$, and finally, by solving $L^{\top}\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$.

IX. `while` Loops and Numerical Convergence: the Bread and Butter of Economic Computation.

A. COMPONENTS:

- Initial guess at the solution
- Iterative steps closer
- Distance metric
- Stopping rule (tolerance of error)

B. NEWTON'S METHOD: the "go-to guy" of numerical optimization and systems of equations

- IDEA: To find the zero of a non-linear function, replace it with a linear approximation to make it simpler.

1. Supply an initial guess x_0 for the zero of a function $f(x)$.

2. Linearize at x_0 $f(x_0) \approx f(x_0) + f'(x_0)(x - x_0)$ and find the zero, to get the updated guess, $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_0)}$.

3. Repeat step 2. until $\|f(x_{k+1})\| < \varepsilon$ for some stopping rule, ε.

- ADVANTAGES: convergence with <u>blazing speed</u>.

- DISADVANTAGE: the algorithm need not converge in general, but there are many ways to overcome this.

Newton's method will be covered more in depth in Computation Notes, Part III.

---

[3]Look up Cholesky Decomposition and LU Decomposition on the Mathworld website for more details.

DIGRESSION ON CONDITION NUMBER: In order to get accuracy on a computer when computing the solution to $A\mathbf{x} = \mathbf{b}$, we need for the solution to be robust to small rounding errors. However, many systems encountered in practice are ill-conditioned, meaning that small changes to $\mathbf{b}$ of size δ lead to large changes in $\mathbf{x}$, relative to δ. We measure this tendency by the following "elasticity":

$$\varepsilon_{\mathbf{x},\mathbf{b}} = \sup_{\|\delta\mathbf{b}\|>0} \frac{\|\delta\mathbf{x}\|/\|\mathbf{x}\|}{\|\delta\mathbf{b}\|/\|\mathbf{b}\|}.$$

Although this provides a useful measure, the above elasticity is difficult to compute. However, the *condition number* of the matrix A provides a least upper bound of $\varepsilon_{\mathbf{x},\mathbf{b}}$:

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|.$$

As it turns out, this is equivalent to computing

$$\kappa(A) = \left| \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} \right|,$$

where $\lambda_{\max}(A)$ and $\lambda_{\min}(A)$ are the maximal and minimal eigenvalues of the matrix A. Thus, for any matrix, the ratio of the extremal eigenvalues will give one an idea of the worst case one can expect. When the condition number is either very small or very large, numerical stability (or lack thereof) is a concern. The command to compute the condition number in MATLAB is `cond(A)`.