Key Concepts: Economic Computation, Part II

Brent Hickman*

Fall, 2009

The purpose of the second section of these notes is to give you some further practice with numerical computation in MATLAB, and also to help you begin to develop a numerical mindset (as opposed to an analytic one).

1 Naive Search vs. Divide and Conquer

In this section we will talk about some simple algorithms for looking up items in a sorted list. Broadly speaking, an *algorithm* is a group of instructions that completely characterize the process of converting a set of inputs into desired outputs. To illustrate this concept, we shall solve a basic search problem. Let $L = \{l_1, l_2, \ldots, l_K\}$ denote a set of ordered objects, where l_k precedes l_{k+1} in the ordering for all k. For example, the list L can be a list of alphabetized words or a set of real numbers ordered from least to greatest. Let l^* denote a single object similar to those in the list L. Our objective is to find the correct placement for l^* in the list L. In other words, we wish to devise an algorithm that accepts inputs L and l^*, and produces an output $k^* \in \mathbb{N}$ such that the list $L^* = \{l_1, \ldots, l_{k^*-1}, l^*, l_{k^*}, \ldots, l_K\}$ is an ordered list.

There are many possible algorithms we could select for producing the desired

*University of Iowa Brent-Hickman@UIowa.Edu

output from our specified inputs. In order to qualify as a valid algorithm, the proposed process need only satisfy two criteria:

1) It must be able to handle arbitrary inputs satisfying the definition of the problem.

2) It must correctly produce the specified output in a finite number of operations.

However, not all algorithms are created equal, and the criterion typically used to rank algorithms is *computational cost*, or the number of operations required for the algorithm to terminate. Moreover, the computational cost of an algorithm is commonly measured in terms of the worst possible case, or in other words, the set of inputs that would maximize the workload required for termination. Note that two different algorithms may differ by their worst cases: a scenario that requires the maximal number of steps for one need not maximize computational burden for another.

For example, consider two alternative algorithms for the search problem: *naive search* and *divide and conquer*. Naive search is the "brute force" approach. In words, it works by simply comparing l^* to each successive element of L, beginning with the first, until one is found that succeeds l^* in the ordering. The index of the first such element is k^*. It is easy to see that the worst case for naive search is some l^* that succeeds all elements of L; this would require exactly K comparisons of l^* with other objects.

Divide and conquer is a more efficient algorithm that takes advantage of the fact that the list L comes pre-sorted. This means that if l^* succeeds l_k for some k, then it must be that l^* also succeeds $l_{k'}$ for any $k' < k$. Similarly, if l^* precedes l_k for some k, then it must be that l^* also precedes $l_{k'}$ for any $k' > k$. Thus, at each step the algorithm divides the list into two equal parts (hence the name), and compares l^* to the first element of the second half of the list. If l^* succeeds that object, then the first half of the list can be discarded. Otherwise, the second half is discarded. This process is repeated until only a single object remains, at which point k^* is the index of that element of the list.

So what is the computational cost of adopting divide and conquer as our solution to the search problem? Note that with each successive step, we decrease the number of possible comparisons which could be made by half. From this fact it follows that the algorithm will always require $\lceil \log_2(K) \rceil$ comparisons, where $\lceil \cdot \rceil$ denotes the ceiling operator that rounds its argument upward. To illustrate the difference between naive search and divide and conquer, suppose that the length K of the list was $1,000,000$. Then the worst case for the former involves performing $1,000,000$ comparisons, whereas the worst case for the latter involves only 20.

Exercise 1 For the purpose of this exercise, let L be a list of K real numbers sorted in increasing order, and let l^* be a real number. Below are two blocks of "pseudo code" (rough drafts of a computer program) for naive search and divide and conquer.
NAIVE SEARCH:

```
DEFINE i*=K+1
for i=1:K
    if L(i)>l*
        RESET i*=i
        BREAK
    END
END
```

DIVIDE AND CONQUER (here, the `floor` function rounds its argument down):

```
DEFINE i_hi = K, i_lo = 0, i* = floor(K/2)
while i_hi ~= i_lo
    if l* <= L(i*)
        RESET i_hi = i*,
        RESET i* = floor((i_lo+i*)/2)
    else
        RESET i_lo = i*, i* = floor((i_hi+i*)/2)
    END
END
```

a. There are two main types of errors that programmers commit when designing a program. The first type is syntactical error, being specific to the particular programming language in use. The other type is algorithmic error, which is a mistake that causes a computer program to violate one of the two criteria above. In other words, an algorithmic error is a mistake that either *(i)* renders a program incapable of handling some relevant inputs or *(ii)* causes to the process to terminate with an incorrect output, or not to terminate at all. Algorithmic errors are very common for beginning programmers, and can often be difficult to identify.

   One of the two blocks of pseudo code above contains an algorithmic error. Identify the error, state whether it violates criterion 1) or 2) above, and propose a solution for the error. (*HINT:* strategically choose some different inputs and check the processes with them. Try defining a list of, say 4 ordered numbers, and pick an l^* that precedes everything on the list, one that succeeds everything, and another that goes somewhere in the middle.)

b. Write a MATLAB program that randomly generates a set of inputs L and l^*. Code up the solution to the search problem using both naive search and divide and

conquer. Devise a block of code to test whether the output is correct, and output the number of iterations to the screen after termination.

2 Finding Zeros of Real-Valued Functions: The Bisection Method

In this section, we return to the idea of numerical convergence that we touched on in Part I. Specifically, we will find the zero of a function by the *bisection algorithm*. The bisection algorithm accepts as inputs a real-valued, monotonic function $f(x)$, and an ordered pair (a_0, b_0), such that $a_0 < b_0$, and $f(x) = 0$ for some $x \in [a_0, b_0]$. To simplify the discussion, we will assume for now that f is a decreasing function. The algorithm computes an approximate zero x^* (the output) by computing a sequence of successively closer guesses. Specifically, it begins with an initial guess of $x_0 = (a_0 + b_0)/2$ and successively updates a, b, x by the following recursion:

If $f(x_k) > 0$ then $a_{k+1} = x_k$, $b_{k+1} = b_k$, and $x_{k+1} = (a_{k+1} + b_{k+1})/2$,

and if $f(x_k) < 0$ $a_{k+1} = a_k$, $b_{k+1} = x_k$, and $x_{k+1} = (a_{k+1} + b_{k+1})/2$,

until a pre-defined stopping rule is satisfied.

In order to complete the definition of the algorithm, we must specify a stopping rule. There are two main options in this case. For some tolerance level, $\tau > 0$ we could terminate the recursion when $|f(x_{k+1})| < \tau$, or alternatively, we could choose to terminate when $b_{k+1} - a_{k+1} < \tau$. One might ask which stopping rule is preferable, and the answer depends on the context of the problem. The former stopping rule is preferable when the primary concern is *(I)* computing an x^* with a functional value very close to zero, and the latter is preferred when the primary concern is *(II)* computing an x^* very close to the true zero. Since one can only approximate the true zero of the function on a computer, targeting one of these criteria does not

automatically ensure that the other will be satisfied too.

It's also important to recognize that the quality of the approximation along either dimension will be affected by the input function f. For example, if f is very steep within a neighborhood of the zero, then a stopping rule of $b_{k+1} - a_{k+1} < \tau$ may not deliver much precision in terms of criterion *(I)*. Alternatively, if the slope and functional values of f are very close to zero on a non-trivial subset of the interval $[a_0, b_0]$, then a stopping rule of $|f(x_{k+1})| < \tau$ may not deliver much precision along criterion *(II)*. Thus, if both criteria are important, or if little is known about the behavior of the input function close to the zero, the programmer may wish to incorporate both stopping rules into the algorithm.

Bisection provides a convenient way of illustrating factors that influence selection of a stopping rule, but these concepts apply to broader settings as well.

Exercise 2 In this exercise you will code up a bisection algorithm in MATLAB, using the composite stopping rule $|f(x_{k+1})| < \tau$ `&&` $b_{k+1} - a_{k+1} < \tau$.

a. Code up a MATLAB script file to find the zeros of the following two functions by the bisection method: $f(x) = -(10^{-10})x$ and $g(x) = (10^{10})x$. Note that f is DECREASING and g is INCREASING. Use the `inline` utility in MATLAB to define f and g. When the algorithm terminates, have your script output to the screen the number of iterations, the approximate zero, and the functional value of the approximate zero.

b. Code up a function file to find the zero of an arbitrary function via the bisection method. The first line of your function file should be the following:
`function [xstar,f0,iter] = bisect0(func,var,direction,tol,a0,b0).`

   In other words, the function name is `bisect0` and the inputs are: `func`, a character string defining a function (as in the syntax for the inline utility); `var`, a string giving the name of the variable in `func`; direction, a string being either `'increasing'` or `'decreasing'` to indicate the type of monotonicity of `func`

(if something else is supplied, then the function should abort and return an error message; type `help error` on the MATLAB command line for more info); `tol`, a tolerance for the stopping rule; `a0`, an initial guess on the lower bound of the search region; and `b0`, an initial guess on the upper bound. The output arguments are `xstar`, the approximate zero; `f0`, the functional value at the approximate zero and `iter`, the number of iterations required for convergence.

Coding this function should be mostly a matter of simply cutting and pasting from your script in part a.

c. Finally, once part b is done, execute the command

```
[xstar,f0,iter] = bisect0('sqrt(exp(1)*pi)+x^3,'x','increasing',tol,-10*c,10*c)
```

several times in MATLAB and experiment with different tolerances (by choosing values for `tol`) and different initial guesses (by choosing different values for $c > 1$). Which one seems to have a greater effect on the number of iterations required to converge under bisection? Is it tolerance or initial guess? Come to class ready to talk about the results.

3 Derivative-Free Optimization: The Golden Search Method

One major application of zero finding in economics is optimization, although bisection is rarely used. In Part I of these notes, we touched on Newton's method, which uses information on derivatives in order to locate the zero of, say a first-order condition. In part III we will cover Newton's method in greater detail but before moving on, we shall start with a simple optimization algorithm that is conceptually similar to bisection and does not require computation of derivatives: *golden search*. This method is useful in cases where there is a need to optimize a continuous univariate objective function with derivatives that are either ill-behaved or intractable.[1]

[1] Although multi-variate, derivative-free optimization is beyond the scope of this introductory course, the interested reader is directed to Miranda and Fackler (2002) for an introduction to the

Like bisection, golden search accepts as inputs a continuous function f and an ordered pair $(\underline{\alpha}, \overline{\alpha})$ defining the bounds on the initial search region. The first step is to evaluate the objective function at two interior points $\alpha < \alpha'$. After comparing the functional values, the sub-optimal interior point is used to replace the nearest endpoint of the search region, and the process is repeated until the length of the search region collapses to a pre-specified tolerance, τ. The algorithm has some unique and attractive characteristics because the interior points are chosen as

$$\alpha = \varphi\underline{\alpha} + (1-\varphi)\overline{\alpha}, \text{ and } \alpha' = (1-\varphi)\underline{\alpha} + \varphi\overline{\alpha},$$

where $\varphi = (\sqrt{5}-1)/2 \approx .62$ is the inverse of the golden ratio, a number famously venerated by ancient Greek philosophers (hence, the name "golden search").

By choosing the interior points in this way, each successive iteration carries over one interior point from the previous iteration, meaning that only one new objective function evaluation is required. This is one of the seemingly magical properties of the golden ratio: by choosing α and α' as golden means of the endpoints, it turns out that α is a golden mean of $\underline{\alpha}$ and α', and also α' is a golden mean of α and $\overline{\alpha}$.

Although derivative-free methods are typically not the first tool of choice in optimization, golden search does have some advantages. First, it requires only that the objective function be continuous, so it can handle some non-differentiable inputs. Second, golden search is guaranteed to converge to a local optimum. This, of course, is subject to the caveat that multiple re-starts (from different initial guesses on the search region) are required if the objective is not unimodal on $[\underline{\alpha}, \overline{\alpha}]$. Third, golden search terminates in a known number of iterations, since at each step the length of the search region contracts by a factor of exactly φ. This means that numerical convergence obtains in $(\log(\tau) - \log(\overline{\alpha} - \underline{\alpha}))\ /\ \log(\varphi)$ iterations.

Exercise 3 Consider the following objective function: $f(x) = -x^2 + 2\pi x - \pi^2$.

Nelder-Mead algorithm, also known as the simplex algorithm.

a. What is the maximizer of this function?

b. How many iterations are required for golden search to converge with a tolerance of $\tau = 10^{-8}$ and an initial search region of $[-10, 10]$?

c. Code up a golden search algorithm in MATLAB to optimize f. Compare your numerical solution and computational cost with your answers to parts a and b.

4 References:

Miranda, Mario J. and Paul L. Fackler. *A*pplied Computational Economics and Finance. MIT Press, 2002.