This chapter presents a variety of bracketing methods for univariate functions, or functions involving a single variable. Bracketing is the process of identifying an interval in which a local minimum lies and then successively shrinking the interval. For many functions, derivative information can be helpful in directing the search for an optimum, but, for some functions, this information may not be available or might not exist. This chapter outlines a wide variety of approaches that leverage different assumptions. Later chapters that consider multivariate optimization will build upon the concepts introduced here.

3.1 Unimodality

Several of the algorithms presented in this chapter assume unimodality of the objective function. A unimodal function $f$ is one where there is a unique $x^*$, such that $f$ is monotonically decreasing for $x \leq x^*$ and monotonically increasing for $x \geq x^*$. It follows from this definition that the unique global minimum is at $x^*$, and there are no other local minima.\(^1\)

Given a unimodal function, we can bracket an interval $[a, c]$ containing the global minimum if we can find three points $a < b < c$, such that $f(a) > f(b) < f(c)$. Figure 3.1 shows an example.

3.2 Finding an Initial Bracket

When optimizing a function, we often start by first bracketing an interval containing a local minimum. We then successively reduce the size of the bracketed interval to converge on the local minimum. A simple procedure (algorithm 3.1) can be used to find an initial bracket. Starting at a given point, we take a step

\(^1\) It is perhaps more conventional to define unimodal functions in the opposite sense, such that there is a unique global maximum rather than a minimum. However, in this text, we try to minimize functions, and so we use the definition in this paragraph.
in the positive direction. The distance we take is a hyperparameter to this algorithm, but the algorithm provided defaults it to $1 \times 10^{-2}$. We then search in the downhill direction to find a new point that exceeds the lowest point. With each step, we expand the step size by some factor, which is another hyperparameter to this algorithm that is often set to 2. An example is shown in figure 3.2. Functions without local minima, such as $\exp(x)$, cannot be bracketed and will cause bracket_minimum to fail.

```
function bracket_minimum(f, x=0; s=1e-2, k=2.0)
    a, ya = x, f(x)
    b, yb = a + s, f(a + s)
    if yb > ya
        a, b = b, a
        ya, yb = yb, ya
        s = -s
    end
    while true
        c, yc = b + s, f(b + s)
        if yc > yb
            return a < c ? (a, c) : (c, a)
        end
        a, ya, b, yb = b, yb, c, yc
        s *= k
    end
end
```

Algorithm 3.1. An algorithm for bracketing an interval in which a local minimum must exist. It takes as input a univariate function $f$ and starting position $x$, which defaults to 0. The starting step size $s$ and the expansion factor $k$ can be specified. It returns a tuple containing the new interval $[a, b]$.  

2 A hyperparameter is a parameter that governs the function of an algorithm. It can be set by an expert or tuned using an optimization algorithm. Many of the algorithms in this text have hyperparameters. We often provide default values suggested in the literature. The success of an algorithm can be sensitive to the choice of hyperparameter.
3.3  Fibonacci Search

Suppose we have a unimodal \( f \) bracketed by the interval \([a, b]\). Given a limit on the number of times we can query the objective function, Fibonacci search (algorithm 3.2) is guaranteed to maximally shrink the bracketed interval.

Suppose we can query \( f \) only twice. If we query \( f \) on the one-third and two-third points on the interval, then we are guaranteed to remove one-third of our interval, regardless of \( f \), as shown in figure 3.3.

We can guarantee a tighter bracket by moving our guesses toward the center. In the limit as \( \epsilon \to 0 \), we are guaranteed to shrink our interval by a factor of two as shown in figure 3.4.
With three queries, we can shrink the interval by a factor of three. We first query \( f \) on the one-third and two-third points on the interval, discard one-third of the interval, and then sample just next to the better sample as shown in figure 3.5.

![Figure 3.5](image)

For \( n \) queries, the interval lengths are related to the Fibonacci sequence: 1, 1, 2, 3, 5, 8, and so forth. The first two terms are one, and the following terms are always the sum of the previous two:

\[
F_n = \begin{cases} 
1 & \text{if } n \leq 2 \\
F_{n-1} + F_{n-2} & \text{otherwise}
\end{cases}
\]  

(3.1)

Figure 3.6 shows the relationship between the intervals. Example 3.1 walks through an application to a univariate function.

![Figure 3.6](image)

The Fibonacci sequence can be determined analytically using Binet’s formula:

\[
F_n = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}
\]  

(3.2)

where \( \phi = (1 + \sqrt{5})/2 \approx 1.61803 \) is the golden ratio.

The ratio between successive values in the Fibonacci sequence is:

\[
\frac{F_n}{F_{n-1}} = \phi \frac{1 - s^{n+1}}{1 - s^n}
\]  

(3.3)

where \( s = (1 - \sqrt{5})/(1 + \sqrt{5}) \approx -0.382 \).
function fibonacci_search(f, a, b, n; ε=0.01)
    s = (1-√5)/(1+√5)
    ρ = 1 / (φ*(1-s^(n+1))/(1-s^n))
    d = ρ*b + (1-ρ)*a
    yd = f(d)
    for i in 1 : n-1
        if i == n-1
            c = ε*a + (1-ε)*d
        else
            c = ρ*a + (1-ρ)*b
        end
        yc = f(c)
        if yc < yd
            b, d, yd = d, c, yc
        else
            a, b = b, c
        end
    end
    ρ = 1 / (φ*(1-s^(n-i+1))/(1-s^(n-i)))
end
return a < b ? (a, b) : (b, a)

Algorithm 3.2. Fibonacci search to be run on univariate function \( f \), with bracketing interval \([a, b]\), for \( n > 1 \) function evaluations. It returns the new interval \((a, b)\). The optional parameter \( ϵ \) controls the lowest-level interval. The golden ratio \( φ \) is defined in `Base.MathConstants.jl`.

3.4 Golden Section Search

If we take the limit for large \( n \), we see that the ratio between successive values of the Fibonacci sequence approaches the golden ratio:

\[
\lim_{n \to \infty} \frac{F_n}{F_{n-1}} = \phi.
\]  

(3.4)

Golden section search (algorithm 3.3) uses the golden ratio to approximate Fibonacci search. Figure 3.7 shows the relationship between the intervals. Figures 3.8 and 3.9 compare Fibonacci search with golden section search on unimodal and non-unimodal functions, respectively.
Consider using Fibonacci search with five function evaluations to minimize
\( f(x) = \exp(x - 2) - x \) over the interval \([a, b] = [-2, 6]\). The first two function
evaluations are made at \( \frac{F_3}{F_6} \) and \( 1 - \frac{F_3}{F_6} \), along the length of the initial bracketing
interval:

\[
\begin{align*}
    f(x^{(1)}) &= f\left(a + (b-a)\left(1 - \frac{F_5}{F_6}\right)\right) = f(1) = -0.632 \\
    f(x^{(2)}) &= f\left(a + (b-a)\frac{F_5}{F_6}\right) = f(3) = -0.282
\end{align*}
\]

The evaluation at \( x^{(1)} \) is lower, yielding the new interval \([a, b] = [-2, 3]\). Two evaluations are needed for the next interval split:

\[
\begin{align*}
    x_{\text{left}} &= a + (b-a)\left(1 - \frac{F_4}{F_5}\right) = 0 \\
    x_{\text{right}} &= a + (b-a)\frac{F_4}{F_5} = 1
\end{align*}
\]

A third function evaluation is thus made at \( x_{\text{left}} \), as \( x_{\text{right}} \) has already been evaluated:

\[ f(x^{(3)}) = f(0) = 0.135 \]

The evaluation at \( x^{(1)} \) is lower, yielding the new interval \([a, b] = [0, 3]\). Two evaluations are needed for the next interval split:

\[
\begin{align*}
    x_{\text{left}} &= a + (b-a)\left(1 - \frac{F_3}{F_4}\right) = 1 \\
    x_{\text{right}} &= a + (b-a)\frac{F_3}{F_4} = 2
\end{align*}
\]

A fourth functional evaluation is thus made at \( x_{\text{right}} \), as \( x_{\text{left}} \) has already been evaluated:

\[ f(x^{(4)}) = f(2) = -1 \]

The new interval is \([a, b] = [1, 3]\). A final evaluation is made just next to the center of the interval at \( 2 + \epsilon \), and it is found to have a slightly higher value than \( f(2) \). The final interval is \([1, 2 + \epsilon]\).
3.4. Golden Section Search

For \( n \) queries of a univariate function we are guaranteed to shrink a bracketing interval by a factor of \( \phi^{n-1} \).

Algorithm 3.3. Golden section search to be run on a univariate function \( f \), with bracketing interval \([a, b]\), for \( n > 1 \) function evaluations. It returns the new interval \((a, b)\). Julia already has the golden ratio \( \phi \) defined. Guaranteeing convergence to within \( \epsilon \) requires \( n = (b - a)/(\epsilon \ln \phi) \) iterations.

```
function golden_section_search(f, a, b, n)
    ρ = φ⁻¹
    d = ρ * b + (1 - ρ)*a
    yd = f(d)
    for i = 1 : n-1
        c = ρ*a + (1 - ρ)*b
        yc = f(c)
        if yc < yd
            b, d, yd = d, c, yc
        else
            a, b = b, c
        end
    end
    return a < b ? (a, b) : (b, a)
end
```
**Figure 3.8.** Fibonacci and golden section search on a unimodal function.

**Figure 3.9.** Fibonacci and golden section search on a nonunimodal function. Search is not guaranteed to find a global minimum.
3.5 Quadratic Fit Search

Quadratic fit search leverages our ability to analytically solve for the minimum of a quadratic function. Many local minima look quadratic when we zoom in close enough. Quadratic fit search iteratively fits a quadratic function to three bracketing points, solves for the minimum, chooses a new set of bracketing points, and repeats as shown in figure 3.10.

Given bracketing points \( a < b < c \), we wish to find the coefficients \( p_1, p_2, \) and \( p_3 \) for the quadratic function \( q(x) \) that goes through \( (a, y_a), (b, y_b), \) and \( (c, y_c) \):

\[
q(x) = p_1 + p_2 x + p_3 x^2 \\
y_a = p_1 + p_2 a + p_3 a^2 \\
y_b = p_1 + p_2 b + p_3 b^2 \\
y_c = p_1 + p_2 c + p_3 c^2
\] (3.5)

In matrix form, we have

\[
\begin{bmatrix}
y_a \\
y_b \\
y_c
\end{bmatrix} =
\begin{bmatrix}
  1 & a & a^2 \\
  1 & b & b^2 \\
  1 & c & c^2
\end{bmatrix}
\begin{bmatrix}
p_1 \\
p_2 \\
p_3
\end{bmatrix}
\] (3.9)

We can solve for the coefficients through matrix inversion:

\[
\begin{bmatrix}
p_1 \\
p_2 \\
p_3
\end{bmatrix} =
\begin{bmatrix}
  1 & a & a^2 \\
  1 & b & b^2 \\
  1 & c & c^2
\end{bmatrix}^{-1}
\begin{bmatrix}
y_a \\
y_b \\
y_c
\end{bmatrix}
\] (3.10)

Our quadratic function is then

\[
q(x) = y_a \frac{(x - b)(x - c)}{(a - b)(a - c)} + y_b \frac{(x - a)(x - c)}{(b - a)(b - c)} + y_c \frac{(x - a)(x - b)}{(c - a)(c - b)}
\] (3.11)

We can solve for the unique minimum by finding where the derivative is zero:

\[
x^* = \frac{1}{2} \frac{y_a (b^2 - c^2) + y_b (c^2 - a^2) + y_c (a^2 - b^2)}{y_a (b - c) + y_b (c - a) + y_c (a - b)}
\] (3.12)

Quadratic fit search is typically faster than golden section search. It may need safeguards for cases where the next point is very close to other points. A basic implementation is provided in algorithm 3.4. Figure 3.11 shows several iterations of the algorithm.
function quadratic_fit_search(f, a, b, c, n)
    ya, yb, yc = f(a), f(b), f(c)
    for i in 1:n-3
        x = 0.5*(ya*(b^2-c^2)+yb*(c^2-a^2)+yc*(a^2-b^2)) / (ya*(b-c) +yb*(c-a) +yc*(a-b))
        yx = f(x)
        if x > b
            c, yc = x, yx
        else
            a, ya, b, yb = b, yb, x, yx
        end
        elseif x < b
            if yx > yb
                a, ya = x, yx
            else
                c, yc, b, yb = b, yb, x, yx
            end
        end
    end
    return (a, b, c)
end

Algorithm 3.4. Quadratic fit search to be run on univariate function \( f \), with bracketing interval \([a, c]\) with \( a < b < c \). The method will run for \( n \) function evaluations. It returns the new bracketing values as a tuple, \((a, b, c)\).

Figure 3.11. Four iterations of the quadratic fit method.
3.6 Shubert-Piyavskii Method

In contrast with previous methods in this chapter, the Shubert-Piyavskii method is a global optimization method over a domain \([a, b]\), meaning it is guaranteed to converge on the global minimum of a function irrespective of any local minima or whether the function is unimodal. A basic implementation is provided by algorithm 3.5.

The Shubert-Piyavskii method requires that the function be Lipschitz continuous, meaning that it is continuous and there is an upper bound on the magnitude of its derivative. A function \(f\) is Lipschitz continuous on \([a, b]\) if there exists an \(\ell > 0\) such that:

\[
|f(x) - f(y)| \leq \ell |x - y| \quad \text{for all } x, y \in [a, b] 
\]  

(3.13)

Intuitively, \(\ell\) is as large as the largest unsigned instantaneous rate of change the function attains on \([a, b]\). Given a point \((x_0, f(x_0))\), we know that the lines \(f(x_0) - \ell(x - x_0)\) for \(x > x_0\) and \(f(x_0) + \ell(x - x_0)\) for \(x < x_0\) form a lower bound of \(f\).

The Shubert-Piyavskii method iteratively builds a tighter and tighter lower bound on the function. Given a valid Lipschitz constant \(\ell\), the algorithm begins by sampling the midpoint, \(x^{(1)} = (a + b)/2\). A sawtooth lower bound is constructed using lines of slope \(\pm \ell\) from this point. These lines will always lie below \(f\) if \(\ell\) is a valid Lipschitz constant as shown in figure 3.12.

Upper vertices in the sawtooth correspond to sampled points. Lower vertices correspond to intersections between the Lipschitz lines originating from each
sampled point. Further iterations find the minimum point in the sawtooth, evaluate the function at that $x$ value, and then use the result to update the sawtooth. Figure 3.13 illustrates this process.

Figure 3.13. Updating the lower bound involves sampling a new point and intersecting the new lines with the existing sawtooth.

The algorithm is typically stopped when the difference in height between the minimum sawtooth value and the function evaluation at that point is less than a given tolerance $\epsilon$. For the minimum peak $(x^{(n)}, y^{(n)})$ and function evaluation $f(x^{(n)})$, we thus terminate if $y^{(n)} - f(x^{(n)}) < \epsilon$.

The regions in which the minimum could lie can be computed using this update information. For every peak, an uncertainty region can be computed according to:

$$\left[ x^{(i)} - \frac{1}{\ell}(y_{\min} - y^{(i)}), x^{(i)} + \frac{1}{\ell}(y_{\min} - y^{(i)}) \right]$$

(3.14)

for each sawtooth lower vertex $(x^{(i)}, y^{(i)})$ and the minimum sawtooth upper vertex $(x_{\min}, y_{\min})$. A point will contribute an uncertainty region only if $y^{(i)} < y_{\min}$. The minimum is located in one of these peak uncertainty regions.

The main drawback of the Shubert-Piyavskii method is that it requires knowing a valid Lipschitz constant. Large Lipschitz constants will result in poor lower bounds. Figure 3.14 shows several iterations of the Shubert-Piyavskii method.


Algorithm 3.5. The Shubert-
Piyavskii method to be run on
univariate function \( f \), with brack-
eting interval \( a < b \) and Lipschitz
constant \( l \). The algorithm runs
until the update is less than the
tolerance \( \epsilon \). Both the best point
and the set of uncertainty intervals
are returned. The uncertainty
intervals are returned as an array
of \((a,b)\) tuples. The parameter \( \delta \)
is a tolerance used to merge the
uncertainty intervals.

```python
struct Pt
    x
    y
end

function _get_sp_intersection(A, B, l)
    t = ((A.y - B.y) - l*(A.x - B.x)) / 2l
    return Pt(A.x + t, A.y - t*l)
end

function shubert_piyavskii(f, a, b, l, \( \epsilon \), \( \delta \)=0.01)
    m = (a+b)/2
    A, M, B = Pt(a, f(a)), Pt(m, f(m)), Pt(b, f(b))
    pts = [A, _get_sp_intersection(A, M, l),
           M, _get_sp_intersection(M, B, l), B]
    \( \Delta \) = Inf
    while \( \Delta > \epsilon \)
        i = argmin([P.y for P in pts])
        P = Pt(pts[i].x, f(pts[i].x))
        \( \Delta \) = P.y - pts[i].y

        P_prev = _get_sp_intersection(pts[i-1], P, l)
        P_next = _get_sp_intersection(P, pts[i+1], l)

        deleteat!(pts, i)
        insert!(pts, i, P_next)
        insert!(pts, i, P)
        insert!(pts, i, P_prev)
    end

    intervals = []
    P_min = pts[2*(argmin([P.y for P in pts[1:2:end]])) + 1]
    y_min = P_min.y
    for i in 2:2:length(pts)
        if pts[i].y < y_min
            dy = y_min - pts[i].y
            x_lo = max(a, pts[i].x - dy/l)
            x_hi = min(b, pts[i].x + dy/l)

            if isempty(intervals) \&\& intervals[end][2] + \( \delta \) ≥ x_lo
                intervals[end] = (intervals[end][1], x_hi)
            else
                push!(intervals, (x_lo, x_hi))
            end
        end
    end

    return (P_min, intervals)
end
```

© 2019 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
Figure 3.14. Nine iterations of the Shubert-Piyavskii method proceeding left to right and top to bottom. The blue lines are uncertainty regions in which the global minimum could lie.
3.7 Bisection Method

The bisection method (algorithm 3.6) can be used to find roots of a function, or points where the function is zero. Such root-finding methods can be used for optimization by applying them to the derivative of the objective, locating where $f'(x) = 0$. In general, we must ensure that the resulting points are indeed local minima.

The bisection method maintains a bracket $[a, b]$ in which at least one root is known to exist. If $f$ is continuous on $[a, b]$, and there is some $y \in [f(a), f(b)]$, then the intermediate value theorem stipulates that there exists at least one $x \in [a, b]$, such that $f(x) = y$ as shown in figure 3.15. It follows that a bracket $[a, b]$ is guaranteed to contain a zero if $f(a)$ and $f(b)$ have opposite signs.

The bisection method cuts the bracketed region in half with every iteration. The midpoint $(a + b)/2$ is evaluated, and the new bracket is formed from the midpoint and whichever side that continues to bracket a zero. We can terminate immediately if the midpoint evaluates to zero. Otherwise we can terminate after a fixed number of iterations. Figure 3.16 shows four iterations of the bisection method. This method is guaranteed to converge within $\epsilon$ of $x^*$ within $\log \left( \frac{|b-a|}{\epsilon} \right)$ iterations, where $\log$ denotes the base 2 logarithm.

Root-finding algorithms like the bisection method require starting intervals $[a, b]$ on opposite sides of a zero. That is, $\text{sign}(f'(a)) \neq \text{sign}(f'(b))$, or equivalently, $f'(a)f'(b) \leq 0$. Algorithm 3.7 provides a method for automatically determining such an interval. It starts with a guess interval $[a, b]$. So long as the interval is invalid, its width is increased by a constant factor. Doubling the interval size is a common choice. This method will not always succeed as shown in figure 3.17. Functions that have two nearby roots can be missed, causing the interval to infinitely increase without termination.

The Brent-Dekker method is an extension of the bisection method. It is a root-finding algorithm that combines elements of the secant method (section 6.2) and inverse quadratic interpolation. It has reliable and fast convergence properties, and...
Algorithm 3.6. The bisection algorithm, where $f'$ is the derivative of the univariate function we seek to optimize. We have $a < b$ that bracket a zero of $f'$. The interval width tolerance is $\epsilon$. Calling \textbf{bisection} returns the new bracketed interval $[a, b]$ as a tuple.

The prime character $'$ is not an apostrophe. Thus, $f'$ is a variable name rather than a transposed vector $f$. The symbol can be created by typing \texttt{\char101prime} and hitting tab.

Algorithm 3.7. An algorithm for finding an interval in which a sign change occurs. The inputs are the real-valued function $f'$ defined on the real numbers, and starting interval $[a, b]$. It returns the new interval as a tuple by expanding the interval width until there is a sign change between the function evaluated at the interval bounds. The expansion factor $k$ defaults to 2.

### 3.8 Summary

- Many optimization methods shrink a bracketing interval, including Fibonacci search, golden section search, and quadratic fit search.

- The Shubert-Piyavskii method outputs a set of bracketed intervals containing the global minima, given the Lipschitz constant.

- Root-finding methods like the bisection method can be used to find where the derivative of a function is zero.

### 3.9 Exercises

**Exercise 3.1.** Give an example of a problem when Fibonacci search is preferred over the bisection method.

**Exercise 3.2.** What is a drawback of the Shubert-Piyavskii method?

**Exercise 3.3.** Give an example of a nontrivial function where quadratic fit search would identify the minimum correctly once the function values at three distinct points are available.

**Exercise 3.4.** Suppose we have \( f(x) = x^2/2 - x \). Apply the bisection method to find an interval containing the minimizer of \( f \) starting with the interval \([0, 1000]\). Execute three steps of the algorithm.

**Exercise 3.5.** Suppose we have a function \( f(x) = (x + 2)^2 \) on the interval \([0, 1]\). Is 2 a valid Lipschitz constant for \( f \) on that interval?

**Exercise 3.6.** Suppose we have a unimodal function defined on the interval \([1, 32]\). After three function evaluations of our choice, will we be able to narrow the optimum to an interval of at most length 10? Why or why not?