

Optimality Conditions for Unconstrained Optimization –

How to Know if You (May) Have Found a Minimum

AE 6310: Optimization for the Design of Engineered Systems

Spring 2017

Dr. Glenn Lightsey

Lecture Notes Developed By Dr. Brian German



The Unconstrained Optimization Problem


Minimize $f(\mathbf{x})$

- ❖ If we wish to maximize $f(\mathbf{x})$, then we can instead minimize $-f(\mathbf{x})$
- ❖ The function $f(\mathbf{x})$ is of unknown form; we have little or no knowledge about its shape
- ❖ $f(\mathbf{x})$ may be expensive to evaluate; we should “query” it as little as possible to reduce the number of “function calls” (queries to the objective function)



Conditions for Optimality

Definition of a Local Solution

- ❖ \mathbf{x}^* is a (weak) local solution to the minimization problem if 
 $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all \mathbf{x} in a neighborhood \mathbf{N} around \mathbf{x}^* .
- ❖ \mathbf{x}^* is a strong local solution to the minimization problem if
 $f(\mathbf{x}^*) < f(\mathbf{x})$ for all \mathbf{x} in a neighborhood \mathbf{N} around \mathbf{x}^* .



Conditions for Optimality

First-Order Necessary Condition

Let $f(\mathbf{x})$ be a function that is once differentiable at \mathbf{x}^* :

- ❖ If \mathbf{x}^* is a local solution to the unconstrained minimization problem, then $\nabla f(\mathbf{x}^*) = \mathbf{0}$.

In other words:

- ❖ $\nabla f(\mathbf{x}^*) = \mathbf{0}$ is necessary for \mathbf{x}^* to be a local solution to the unconstrained minimization problem



Conditions for Optimality

Second-Order Optimality Conditions

Let $f(\mathbf{x})$ be a function that is twice differentiable at \mathbf{x}^* :

- ❖ **Necessity:** If \mathbf{x}^* is a local solution to the unconstrained minimization problem, then $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $H(\mathbf{x}^*)$ is positive semi-definite.
- ❖ **Sufficiency:** If $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $H(\mathbf{x}^*)$ is positive definite, then \mathbf{x}^* is a strong local solution to the unconstrained minimization problem.



Conditions for Optimality

In other words:

- ❖ **Necessity:** $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $H(\mathbf{x}^*)$ being positive semi-definite are necessary for \mathbf{x}^* to be a local solution to the unconstrained minimization problem.
- ❖ **Sufficiency:** $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $H(\mathbf{x}^*)$ being positive definite are sufficient for \mathbf{x}^* to be a strong local solution to the unconstrained minimization problem.



Convexity

The function $f(\mathbf{x})$ is said to be **convex** if and only if for every two points \mathbf{x}_1 and \mathbf{x}_2 in the domain of $f(\mathbf{x})$ the function satisfies,

$$f(\lambda \mathbf{x}_1 + (1 - \lambda) \mathbf{x}_2) \leq \lambda f(\mathbf{x}_1) + (1 - \lambda) f(\mathbf{x}_2)$$

for all $\lambda \in [0,1]$.



The Global Optimum of a Convex Functions

Let $f(\mathbf{x})$ be convex. If \mathbf{x}^* is a local solution to the unconstrained minimization problem, then \mathbf{x}^* is a **global** solution to the unconstrained minimization problem.



Introduction to Optimization Algorithms

AE 6310: Optimization for the Design of Engineered Systems

Dr. Brian German



TANSTAAFL (in Optimization)

- ❖ The “no free lunch” theorem (Wolpert 1997) states that no optimization algorithm performs better than any other *when averaged over all possible problems*.
- ❖ What this really means: Optimization algorithms are specialized to work well on particular types of problems. There is no “universally good” optimization algorithm.
- ❖ Two conclusions:
 1. To be a good user of optimization, you must learn to match features of algorithms to features of problems.
 2. There are LOTS of different optimization algorithms.



What *is* an optimization algorithm?

- ❖ What defines an optimization algorithm?
- ❖ If I describe an algorithm to you, what features would lead you to classify it as an “optimization” algorithm?



What *is* an optimization algorithm?

- ❖ An optimization algorithm is one that evaluates a sequence of designs (\mathbf{x}) with the goal of finding good designs.
- ❖ Most optimization algorithms work by iteratively asking/answering the question “What design(s) should we try next?”
- ❖ (Some optimization algorithms are not iterative. For example, the algorithm “Evaluate 1000 random designs; Pick the best one.” does not involve any iteration. Is this really an “optimization algorithm”?)



What *is* an optimization algorithm?

❖ Without trying to be too precise, *most* iterative optimization algorithms follow these general steps:

1. Begin at a “starting point”, i.e. some design (\mathbf{x}). This becomes the (first) “current point”.
 - Most algorithms provide no guidance on what this starting point should be; rather it is something you bring to the problem as prior knowledge.
 - Often, the starting point is random (e.g. no prior knowledge).
2. Gather some information about the current state of knowledge. This may include (but is not limited to)
 - Information about the current point itself
 - Information about derivatives at the current point (often calculated with finite difference)
 - Information about previously visited points, and/or the path to get to the current point



What *is* an optimization algorithm?

3. Based on the information gathered, select a subset of the design space, to search for a “better” design than the current point.
 - This subset may be:
 - A “search direction” (we call these algorithms “**Line Search**” techniques)
 - A “neighborhood” (we call these algorithms “**Trust Region**” techniques)
 - ...or something else. But *most* algorithms use a line search or a trust region.
 - We are searching for a “better” point than our current point, but this doesn’t strictly mean “a point that improves the objective function”. Some algorithms purposely move to a worse objective value to learn something new about the design space. So “better” is defined by each individual algorithm.
 - ...but for *most* algorithms, “better” does simply mean “a point that improves the objective function”



What *is* an optimization algorithm?

4. Search this subset for a better point. Stop searching when the information you have obtained during the search suggests that:
 1. You have found the best possible point in the subset, or...
 2. You have found a good point, and don't believe that further searching (in this subset) will be worthwhile, or...
 3. No point in the subset is better than the current point.
5. Move to the new "better point" so that it becomes the "current point". **Go back to Step 2**, or quit searching entirely if:
 1. You have found a point that is "good enough".
 2. You believe further searching will not yield any better points.
 3. You have run out of time or resources.
 4. Etc...



So how do we make an optimization algorithm?

Based on this process, we can identify some questions that must be answered to define an optimization algorithm:

- ❖ How does the algorithm select the starting point?
- ❖ What information does that algorithm use to identify a local subset to search?
 - Derivatives around current point? Derivatives around previous points? Other information about previous points? Randomness? Sequences?
- ❖ What “shape” is the subset to be searched? Line? Trust Region? Other?
 - Also, how “big” is the subset? How far from the current point are we willing to look?
- ❖ ...



So how do we make an optimization algorithm?

- ❖ ...
- ❖ What technique is used to search this subset?
 - (This search is *itself* an entire optimization algorithm)
- ❖ How do we define “better” points in this subset? Based on the objective function, or something else?
 - If our problem has constraints, how do we enforce the constraints during this search?
- ❖ When do we stop the algorithm? How do we know when it is finished? If we don't like the result, is there a strategy for restarting with new conditions?



Unconstrained Optimization: Line Search Techniques

AE 6310: Optimization for the Design of Engineered Systems

Dr. Brian German



Types of Unconstrained Optimization Algorithms

Direct search methods

- ❖ Methods for which **derivatives are not used in the search**
- ❖ Search direction is based on a sequence, pattern, or is random. E.g. "Search x_1 direction first. Then search x_2 direction,..."

Line search methods

- ❖ Methods in which the search is conducted along successive lines in the design space
- ❖ Typically **use derivatives in the search**, *but not always*

Trust region methods

- ❖ Methods in which the search is conducted by locally approximating the objective function as being quadratic or linear.
- ❖ The name "trust region" refers to how far from the current point this approximation is valid.



Line Search Methods

- ❖ A line search involves two general steps:
 1. Select a “search direction”
 2. Select the distance to move along this direction
- ❖ This lecture primarily focuses on techniques for determining the distance to move along the search direction.
- ❖ The selection of the direction itself is discussed later when we implement line searches in “named” optimization algorithms.



Ingredients of a Line Search Method

Line search methods are based on an iterative procedure of the form,

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha^* \mathbf{s}_k$$

where,

$\mathbf{x}_k \equiv$ new design variable vector

$\mathbf{x}_{k-1} \equiv$ previous design variable vector

$\mathbf{s}_k \equiv$ a vector in a *descent direction* that defines the line being searched. Finding \mathbf{s}_k is the *direction-finding problem*.

$\alpha^* \equiv$ a scalar indicating the distance we should move to (approximately) minimize the function along the line. Finding α^* is the task of the *line search* itself.



Ingredients of a Line Search Method

To define an algorithm for a particular line search method, we therefore need to specify how the method will accomplish each of the following tasks:

- ❖ Choose an initial point in the design space, \mathbf{x}_0
- ❖ Select a *descent direction*, \mathbf{s}_k , i.e. a direction along which the function value decreases.
- ❖ Determine the α^* that will minimize $f(\mathbf{x})$ (approximately) along the line defined by \mathbf{s}_k
- ❖ Decide when the process has converged to an acceptable solution in order to stop iterating



Types of Line Search Methods

Line search methods are categorized by the quality of the local approximation of the function that they use in the direction-finding problem to determine \mathbf{s}_k :

Zeroth-order methods

- ❖ Use only values of the function $f(\mathbf{x})$ itself
- ❖ Some consider zeroth-order methods to be direct search methods

First-order methods

- ❖ Use values of $f(\mathbf{x})$ and $\nabla f(\mathbf{x})$
- ❖ May use approximations of $H(\mathbf{x})$ but are still of first-order accuracy

Second-order methods

- ❖ Use values of $f(\mathbf{x})$, $\nabla f(\mathbf{x})$, and $H(\mathbf{x})$



The Line Search Itself: Estimating α^*

- ❖ Estimating α^* requires a search along the line defined by the direction \mathbf{s}_k to find an estimate of the minimum along that line
- ❖ The advantage of this approach is that it is a 1-D search, which is much easier than an n-D search

We will discuss two approaches for finding α^* :

- ❖ Polynomial Approximations
- ❖ Golden Section Method



Polynomial Approximations

The idea is that we will approximate the function along the line defined by \mathbf{s}_k with a polynomial in α :

$$f(\mathbf{x}) \approx \tilde{f}(\mathbf{x}) = b_0 + b_1\alpha + b_2\alpha^2 + \cdots + b_n\alpha^n$$

We are typically interested in “reasonable” but inexact approximations of the minimum along the search line. Therefore, the most common approximations are quadratics and cubics, which provide a nice balance between accuracy and efficiency.



2-Point Quadratic Approximation

Let's look at a quadratic approximation in which we know information at two points:

$$f(\mathbf{x}) = f(\mathbf{x}_{k-1} + \alpha \mathbf{s}_k) \approx \tilde{f}(\mathbf{x}) = b_0 + b_1 \alpha + b_2 \alpha^2$$

$$\frac{df(\mathbf{x})}{d\alpha} = \frac{df(\mathbf{x})}{d\mathbf{x}} \frac{d\mathbf{x}}{d\alpha} = [\nabla f(\mathbf{x}_{k-1})]^T \mathbf{s}_k \approx \frac{d\tilde{f}(\mathbf{x})}{d\alpha} = b_1 + 2b_2 \alpha$$

- ❖ At point 1, corresponding to α_1 , we know both $\left. \frac{df}{d\alpha} \right|_1$ and f_1
- ❖ At point 2, corresponding to α_2 , we know f_2



2-Point Quadratic Approximation

Substituting the known values gives,

$$\left. \frac{df}{d\alpha} \right|_1 = b_1 + 2b_2\alpha_1$$

$$f_1 = b_0 + b_1\alpha_1 + b_2\alpha_1^2$$

$$f_2 = b_0 + b_1\alpha_2 + b_2\alpha_2^2$$

With α_1 , $\left. \frac{df}{d\alpha} \right|_1$, f_1 , α_2 , and f_2 known, we need to solve for the coefficients b_0 , b_1 , and b_2 to find the polynomial approximation.



2-Point Quadratic Approximation

Subtracting the two function values and dividing by $(\alpha_2 - \alpha_1)$,

$$f_2 - f_1 = b_1(\alpha_2 - \alpha_1) + b_2(\alpha_2^2 - \alpha_1^2)$$

$$\frac{f_2 - f_1}{\alpha_2 - \alpha_1} = b_1 + b_2(\alpha_2 + \alpha_1)$$

But,

$$b_1 = \left. \frac{df}{d\alpha} \right|_1 - 2b_2\alpha_1$$

So,

$$\frac{f_2 - f_1}{\alpha_2 - \alpha_1} = \left. \frac{df}{d\alpha} \right|_1 + b_2(\alpha_2 - \alpha_1)$$



2-Point Quadratic Approximation

Solving for b_2 gives

$$b_2 = \frac{(f_2 - f_1)/(\alpha_2 - \alpha_1) - \left. \frac{df}{d\alpha} \right|_1}{(\alpha_2 - \alpha_1)}$$

With b_2 known, we can then find b_1 and b_0 directly as,

$$b_1 = \left. \frac{df}{d\alpha} \right|_1 - 2b_2\alpha_1$$

$$b_0 = f_1 - b_1\alpha_1 - b_2\alpha_1^2$$



2-Point Quadratic Approximation

The critical point of \tilde{f} along the search line is then found as,

$$\frac{d\tilde{f}}{d\alpha} = b_1 + 2b_2\alpha = 0$$

$$\Rightarrow \alpha_{cr} = \frac{-b_1}{2b_2}$$

If $b_2 > 0$, the critical point is a minimum.

If $b_2 < 0$, the critical point is a maximum.

If the critical point is a minimum, then $\alpha^* = \alpha_{cr}$. Since we search in a descent direction, we should hopefully find a minimum.



3-Point Quadratic Approximation

Consider again our quadratic approximation:

$$\tilde{f}(\boldsymbol{x}) = b_0 + b_1\alpha + b_2\alpha^2$$

$$\frac{d\tilde{f}(\boldsymbol{x})}{d\alpha} = b_1 + 2b_2\alpha$$

Presume that instead of knowing the function value at 2 points and the derivative at 1 point, we know the function value at 3 points, i.e.

$$(\alpha_1, f_1), (\alpha_2, f_2), (\alpha_3, f_3)$$



3-Point Quadratic Approximation

We can then solve for the coefficients as,

$$b_2 = \frac{(f_3 - f_1)/(\alpha_3 - \alpha_1) - (f_2 - f_1)/(\alpha_2 - \alpha_1)}{(\alpha_3 - \alpha_2)}$$

$$b_1 = \frac{(f_2 - f_1)}{(\alpha_2 - \alpha_1)} - b_2(\alpha_1 + \alpha_2)$$

$$b_0 = f_1 - b_1\alpha_1 - b_2\alpha_1^2$$



Other Polynomial Approximations

The Vanderplaats textbook presents several other approximations in Chapter 2. These include:

- ❖ 3-point cubic
- ❖ 4-point cubic
- ❖ 1-point linear
- ❖ 2-point linear

You may wish to review these approximations.



Selecting the Points for Defining the Polynomial

We now know how to define a polynomial once we are given several points along a search direction. But how do we select the points, i.e. values of α_1, α_2 , etc.?

Things to note:

- ❖ We start the line search at the point \mathbf{x}_k at which $\alpha = 0$. It is convenient to associate α_1 with this starting point. This is also the point at which we may know the gradient.
- ❖ For 3-point methods, presuming that we number the α_i in order of increasing α , we need to select α_2 and α_3 such that $f_2 < \min(f_1, f_3)$. This is called bracketing the minimum.



Bracketing the Minimum

Presuming α_1 is fixed and corresponds to the starting point \mathbf{x}_k , we can use a bracketing algorithm to select points α_2 and α_3 .

Let's begin by making the following presumptions:

❖ The function is **unimodal** along the search direction

❖ $\left. \frac{df}{d\alpha} \right|_1 < 0$, i.e. the search is in a **descent direction**

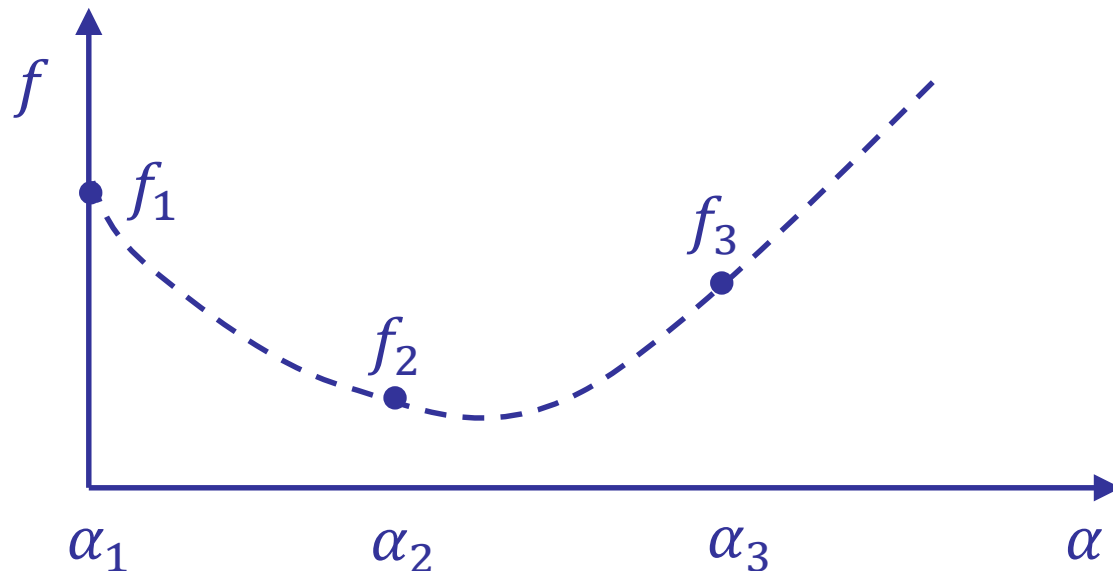
These conditions imply that a minimum defined by α^* and f^* exists along the search direction.



Bracketing the Minimum

Presume that we select α_2 and α_3 and evaluate the corresponding values of f_1 , f_2 , and f_3 . Three possibilities exist.

1. Minimum already bracketed: $f_2 < \min(f_1, f_3)$



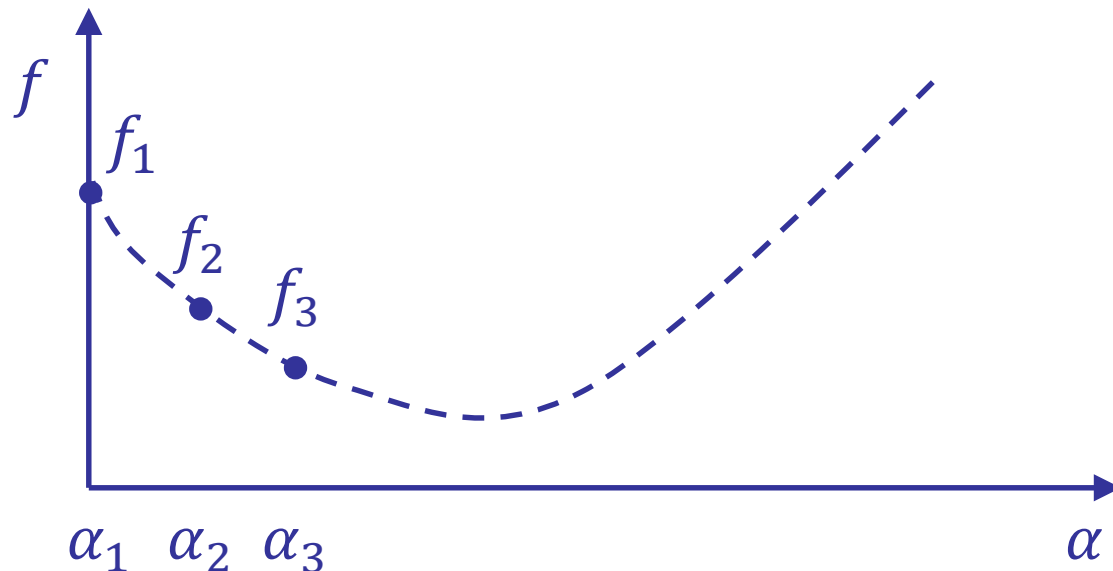
Adapted from notes
by Prof. Rob McDonald of
CalPoly San Luis Obispo



Bracketing the Minimum

Presume that we select α_2 and α_3 and evaluate the corresponding values of f_1 , f_2 , and f_3 . Three possibilities exist.

2. Interval too small: $f_3 < \min(f_1, f_2)$



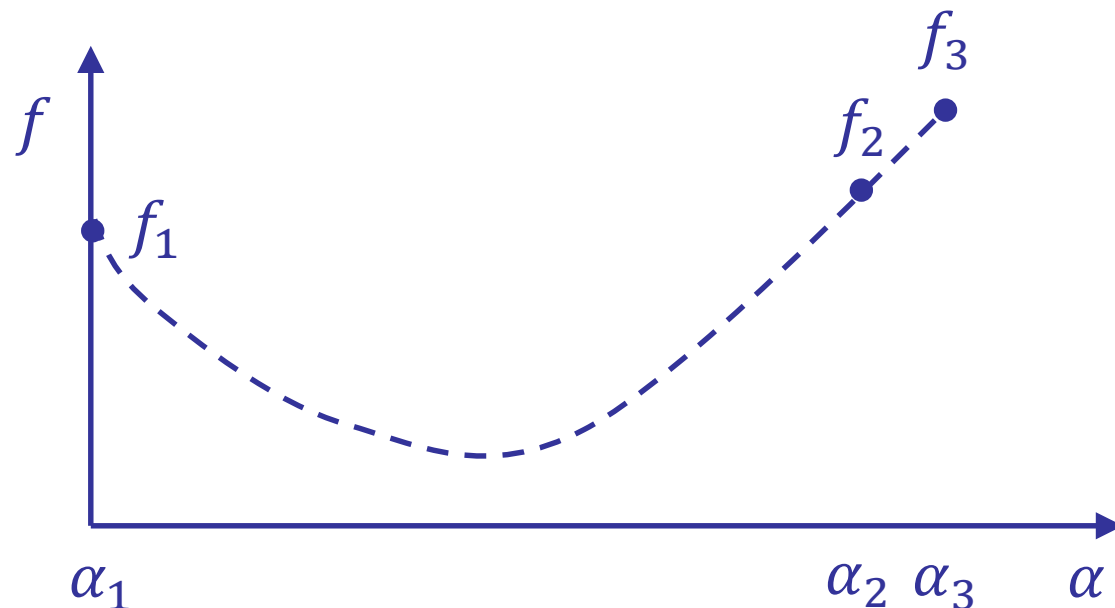
Adapted from notes
by Prof. Rob McDonald of
CalPoly San Luis Obispo



Bracketing the Minimum

Presume that we select α_2 and α_3 and evaluate the corresponding values of f_1 , f_2 , and f_3 . Three possibilities exist.

3. Interval too large: $f_1 < \min(f_2, f_3)$

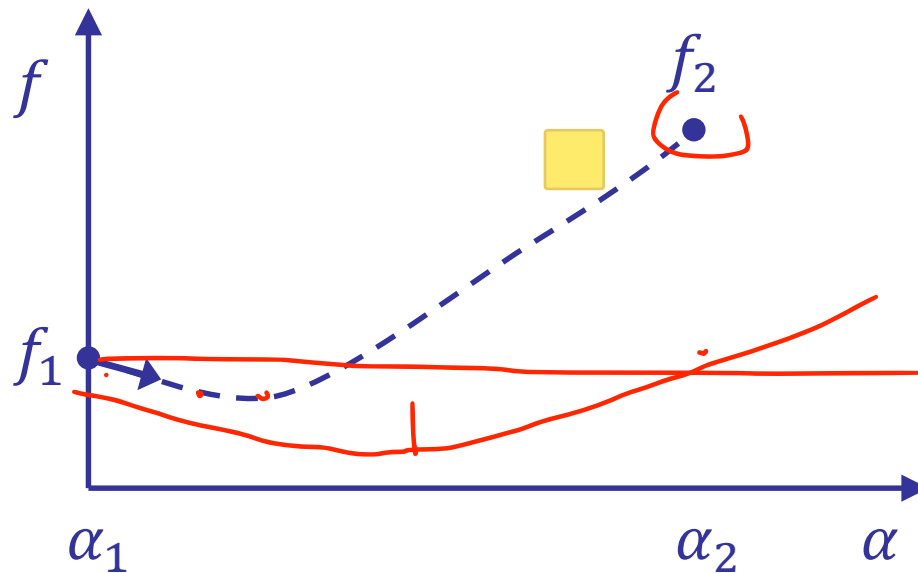


Adapted from notes
by Prof. Rob McDonald of
CalPoly San Luis Obispo



Bracketing the Minimum

How does the approach change if we consider a 2-point approximation?



Adapted from notes
by Prof. Rob McDonald of
CalPoly San Luis Obispo



Bracketing the Minimum

Let's now presume that we work with a fixed interval width,

$$\Delta\alpha = \alpha_3 - \alpha_2 = \alpha_2 - \alpha_1$$

Here's an approach to adjusting $\Delta\alpha$ based on the initial points:

- ❖ If the minimum is **already bracketed**, $\Delta\alpha$ is fine as it is. Proceed.
- ❖ If the interval is **too small**, *multiply* $\Delta\alpha$ by 2 and try again
- ❖ If the interval is **too large**, *divide* $\Delta\alpha$ by 2 and try again

When bracketing the minimum, you may need to shrink/grow the interval several times. However, if you start by shrinking it, you never need to grow it, and vice versa.

Note that approaches other than multiplying/dividing $\Delta\alpha$ by 2 are possible.



Bracketing the Minimum

The presumption that the function is unimodal along the search direction is not critical. If the function is multimodal, the algorithm presented in the previous slides will bracket one of the local minima.

A more significant problem occurs if the function is unbounded in the search direction, e.g. a linear function. Special checks or limits to the number of bracketing attempts must be included to handle this issue.

Adapted from notes
by Prof. Rob McDonald of
CalPoly San Luis Obispo



The Golden Section Method

The **Golden Section Method** is an approach that successively refines the bracketing of the minimum in order to converge to an estimate of the minimum.

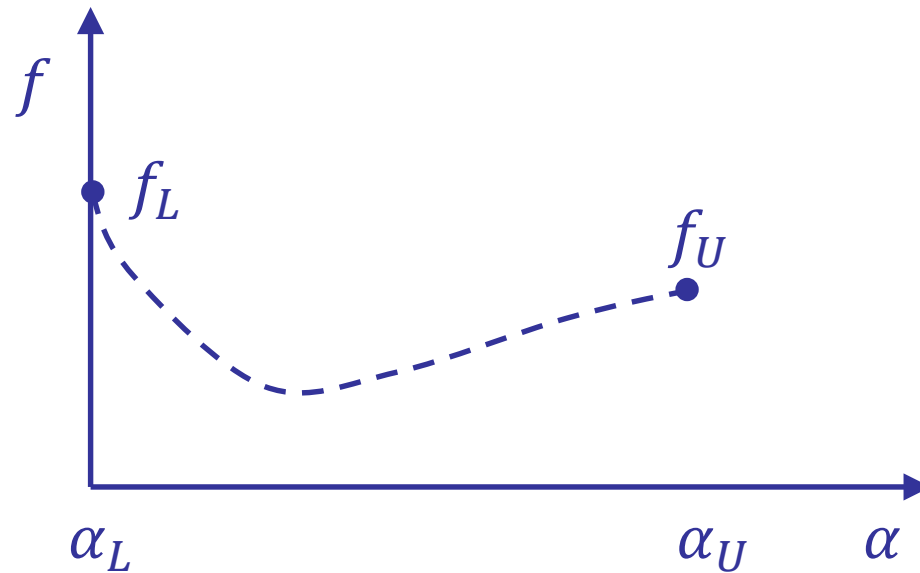
The method takes its name from the way it selects new points, which is related to the “golden section” or “golden ratio”:

$$\frac{1+\sqrt{5}}{2} \approx 1.61803$$



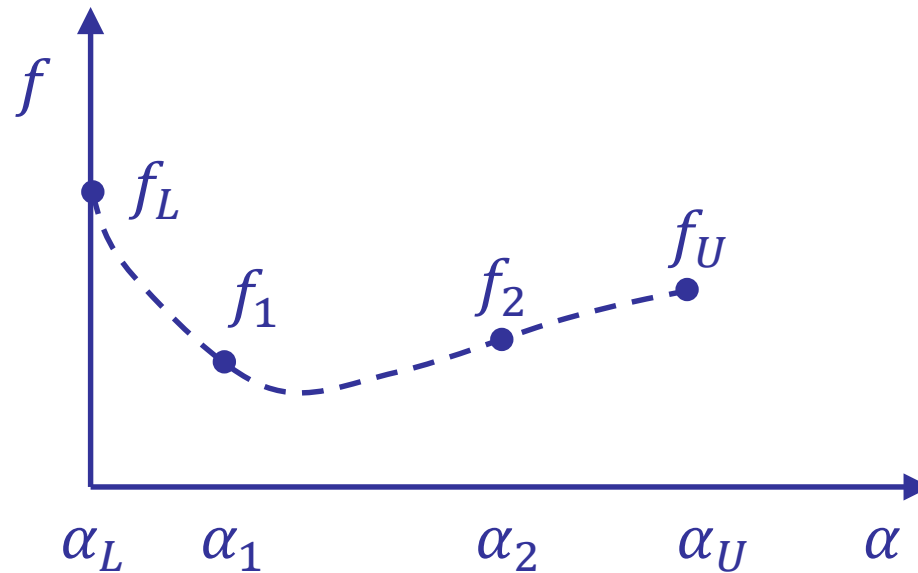
The Golden Section Method

To understand how the GSM works, imagine that we begin with a bracketed minimum. (Presume that we know an intermediate point that is lower than f_L and f_U)



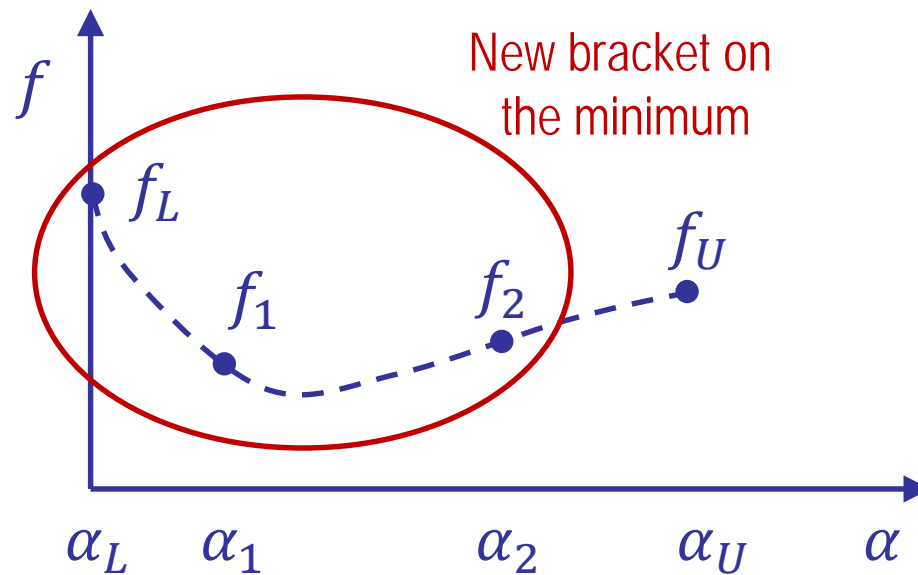
The Golden Section Method

To refine the bracketing, we add two intermediate points $\alpha_1 < \alpha_2$ and compute the function values at these points:



The Golden Section Method

One of these two new points will define a new bracket for the minimum. In this case, since $f_1 < \min(f_L, f_2)$, the point α_1 refined the bracket. This implies α_U is no longer needed and can be discarded. This reduces the size of our search region.



The Golden Section Method

Next, we throw away the old bound that no longer brackets the minimum (in this case α_U) and relabel the points by “shifting” the labels of α_1 and α_2 as follows:

❖ If we discard α_U : $\alpha_2 \rightarrow \alpha_U, \quad \alpha_1 \rightarrow \alpha_2$

❖ If we discard α_L : $\alpha_1 \rightarrow \alpha_L, \quad \alpha_2 \rightarrow \alpha_1$

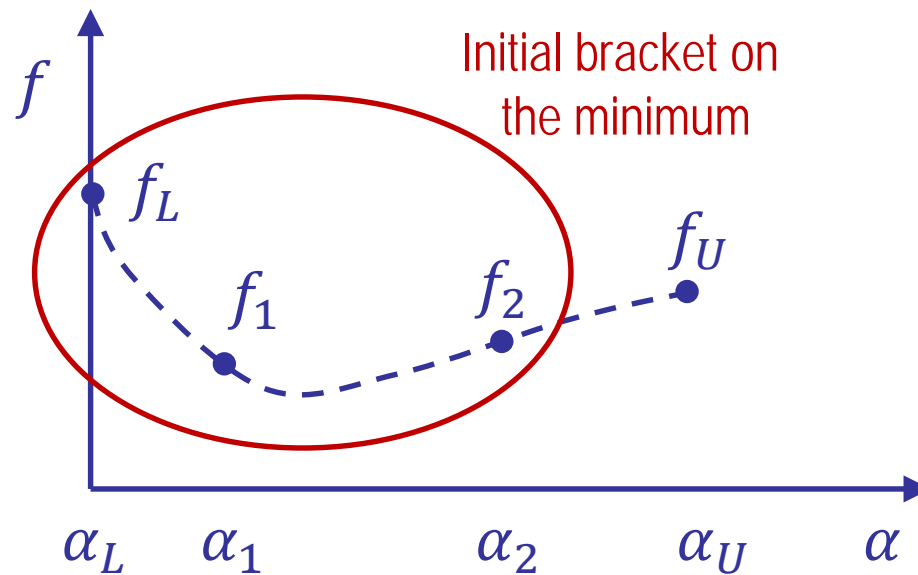
Finally, we add a new point to replace the value of α that is now missing, i.e. either α_1 or α_2



The Golden Section Method

Example of discarding the old bound, relabeling the points, and adding a new point:

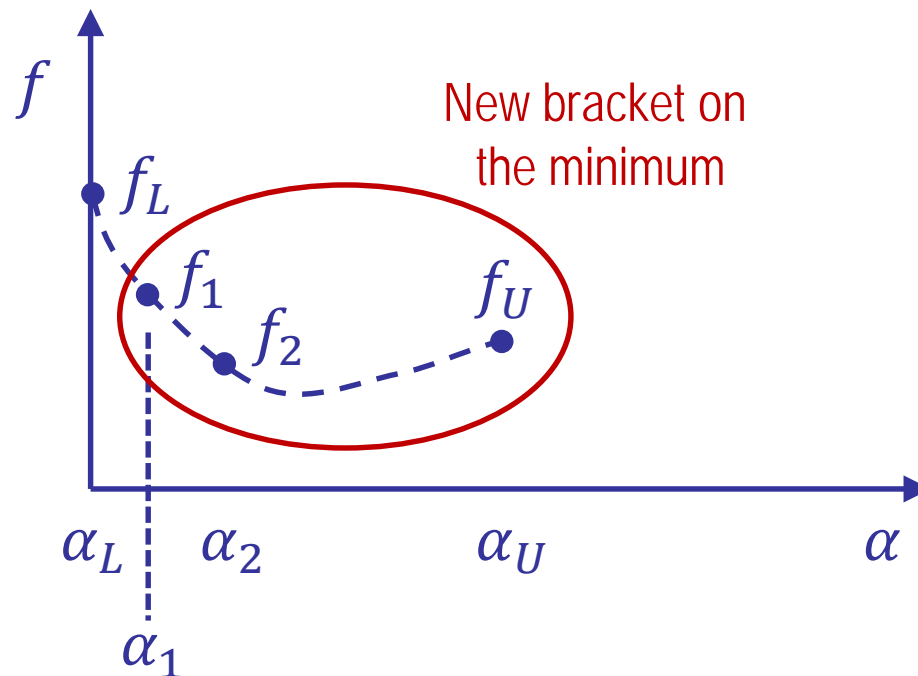
Initial points and their labeling:



The Golden Section Method

Example of discarding the old bound, relabeling the points, and adding a new point:

New points and their labeling:



The Golden Section Method

You can imagine how we can continue iterating this approach to successively refine the bracket and “home in” on the minimum.

The only remaining question is how to specify an appropriate spacing rule to define at which specific locations along the α axis to add in the new points α_1 , α_2 , etc.

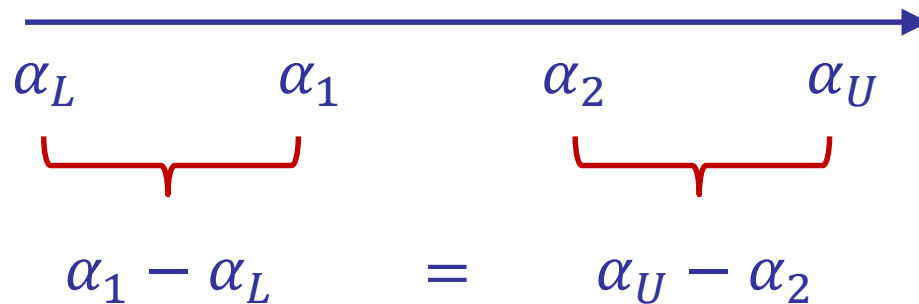
As mentioned earlier, a very effective spacing rule will turn out to be based on the “golden section”, from which the method takes its name.



The Golden Section Method

“The most efficient algorithm is the one that reduces the bounds by the same fraction on each iteration.” (Vanderplaats, p. 59)

In order to achieve this consistent fractional reduction, we first decide to space α_1 and α_2 such that $\alpha_1 - \alpha_L = \alpha_U - \alpha_2$:

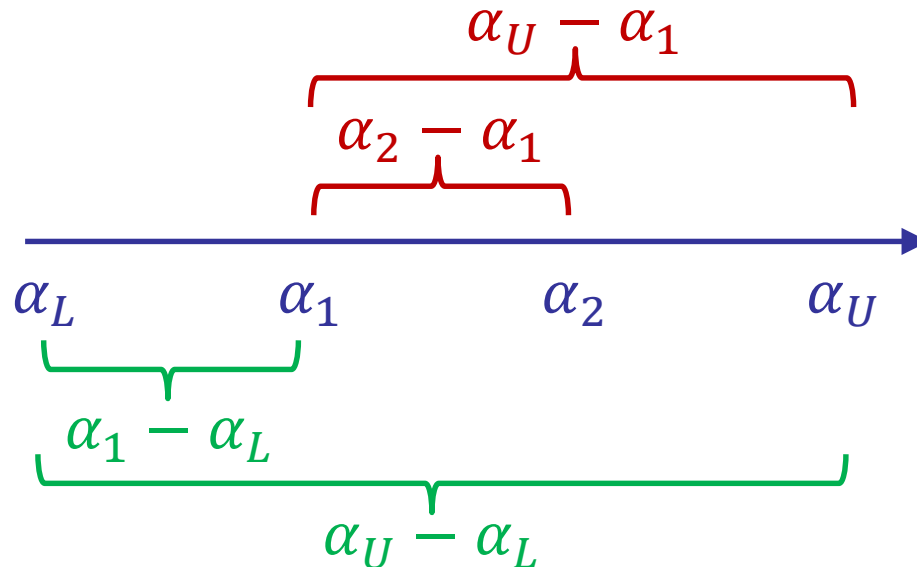


The Golden Section Method

Next, we require,

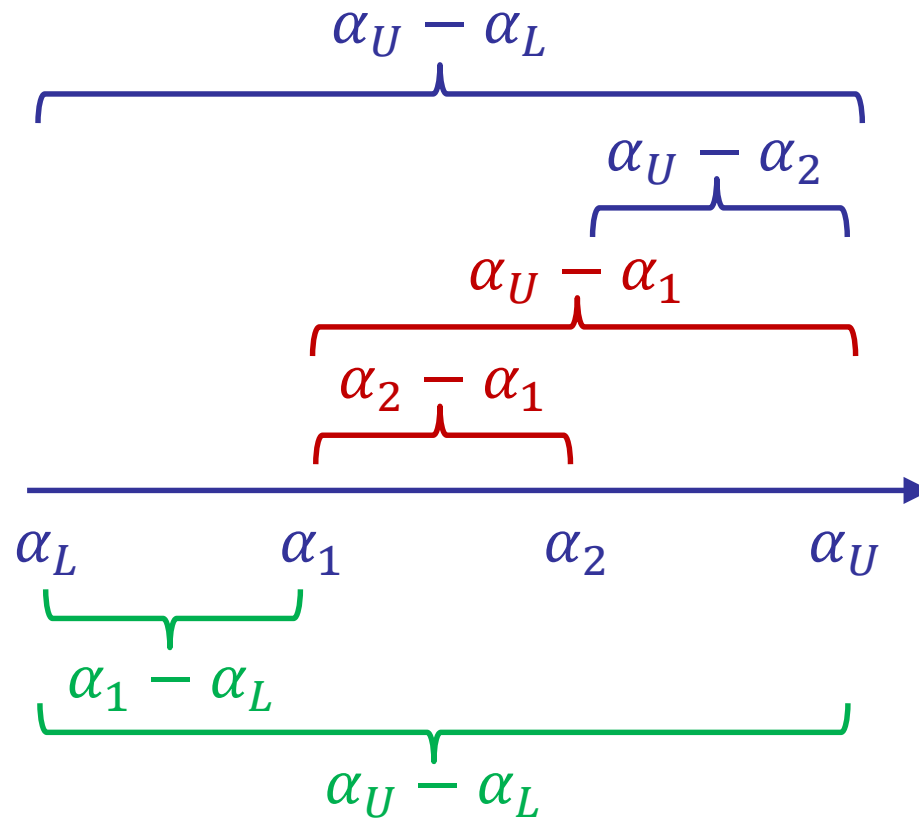
$$\frac{\alpha_1 - \alpha_L}{\alpha_U - \alpha_L} = \frac{\alpha_2 - \alpha_1}{\alpha_U - \alpha_1}$$

Why? Let's look at it graphically...



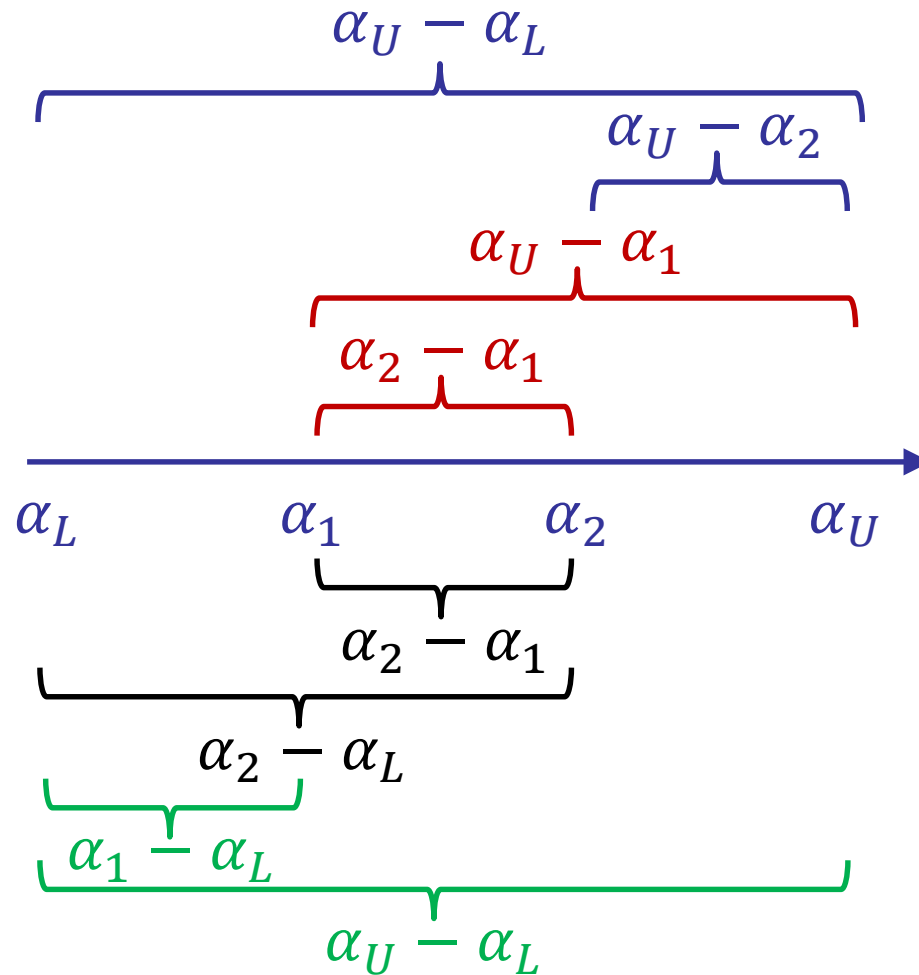
The Golden Section Method

But, since we also required that $\alpha_1 - \alpha_L = \alpha_U - \alpha_2 \dots$



The Golden Section Method

... and since $\alpha_U - \alpha_1 = \alpha_2 - \alpha_L$ (see if you can show this):



The Golden Section Method

The two relations that we have specified therefore guarantee that no matter which bound we discard, we always eliminate the same fraction of the interval.



The Golden Section Method

Let's now apply these relations to solve for values of α_1 and α_2 .

First, presume that we have normalized our bounds on α such that $\alpha_L = 0$ and $\alpha_U = 1$. Applying the ratio rule then results in,

$$\frac{\alpha_1 - \alpha_L}{\alpha_U - \alpha_L} = \frac{\alpha_2 - \alpha_1}{\alpha_U - \alpha_1} \Rightarrow \alpha_1 = \frac{\alpha_2 - \alpha_1}{1 - \alpha_1}$$

But, $\alpha_1 - \alpha_L = \alpha_U - \alpha_2 \Rightarrow \alpha_1 = 1 - \alpha_2 \Rightarrow \alpha_2 = 1 - \alpha_1$,
so,

$$\alpha_1 = \frac{1 - 2\alpha_1}{1 - \alpha_1}$$



The Golden Section Method

We can rearrange this expression as,

$$\alpha_1 = \frac{1 - 2\alpha_1}{1 - \alpha_1} \Rightarrow \alpha_1^2 - 3\alpha_1 + 1 = 0$$

Solving for the roots of the quadratic gives,

$$\alpha_1 = \frac{3 \pm \sqrt{5}}{2} \approx 0.38197, \quad 2.61803$$



The Golden Section Method

The second root is meaningless because it is outside the interval, so we choose the first root,

$$\alpha_1 = \frac{3 - \sqrt{5}}{2} \approx 0.38197$$

$$\alpha_2 = 1 - \alpha_1 = \frac{-1 + \sqrt{5}}{2} \approx 0.61803$$

This implies that we obtain the Golden Section ratio as follows:

$$\frac{\alpha_2}{\alpha_1} = \frac{1 + \sqrt{5}}{2} \approx 1.61803$$



The Golden Section Method

We can now apply this result for the normalized bounds on α to our original (dimensional) bounds on α as follows.

Let $\tau = 0.38197$, then:

$$\alpha_1 = (1 - \tau)\alpha_L + \tau\alpha_U$$

$$\alpha_2 = \tau\alpha_L + (1 - \tau)\alpha_U$$



Convergence of the Golden Section Method

To examine the convergence of the Golden Section Method, consider a relative tolerance, ε , defined as,

$$\varepsilon = \frac{\Delta\alpha}{\alpha_{U,0} - \alpha_{L,0}}$$

where

$\alpha_{U,0} \equiv$ the initial upper bound

$\alpha_{L,0} \equiv$ the initial lower bound

$\Delta\alpha \equiv$ the difference $\alpha_U - \alpha_L$ at each iteration



Convergence of the Golden Section Method

Since the interval is reduced in size by the fraction $\tau = 0.38197$ each iteration,

$$\varepsilon = (1 - \tau)^{N-3}$$

where N is the total number of function calls. The “-3” in the exponent accounts for the function calls in the initial step.

We can now solve for the number of function calls, N needed to achieve a particular tolerance, ε , as,

$$N = \frac{\ln \varepsilon}{\ln(1 - \tau)} + 3$$



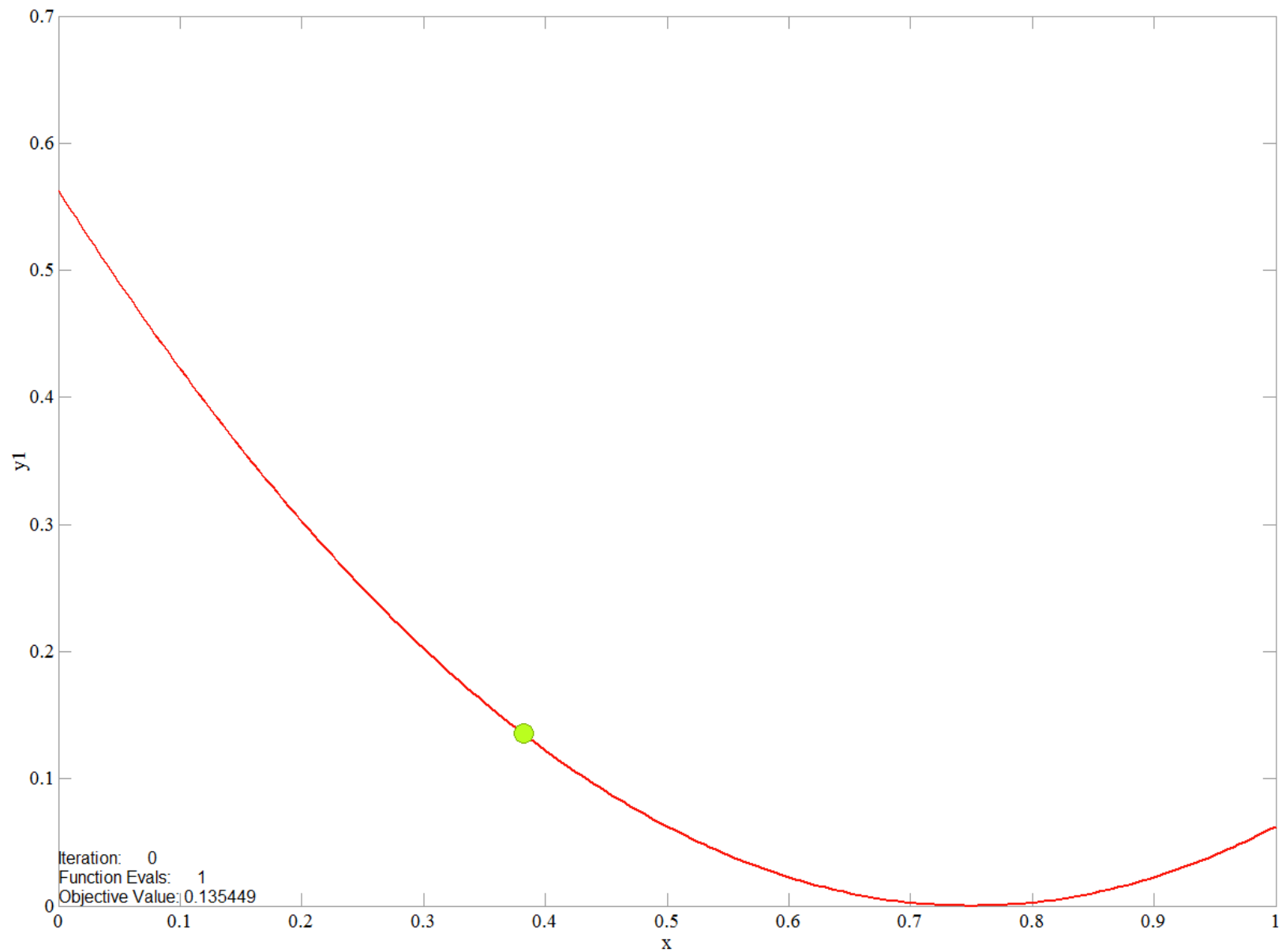
Convergence of the Golden Section Method

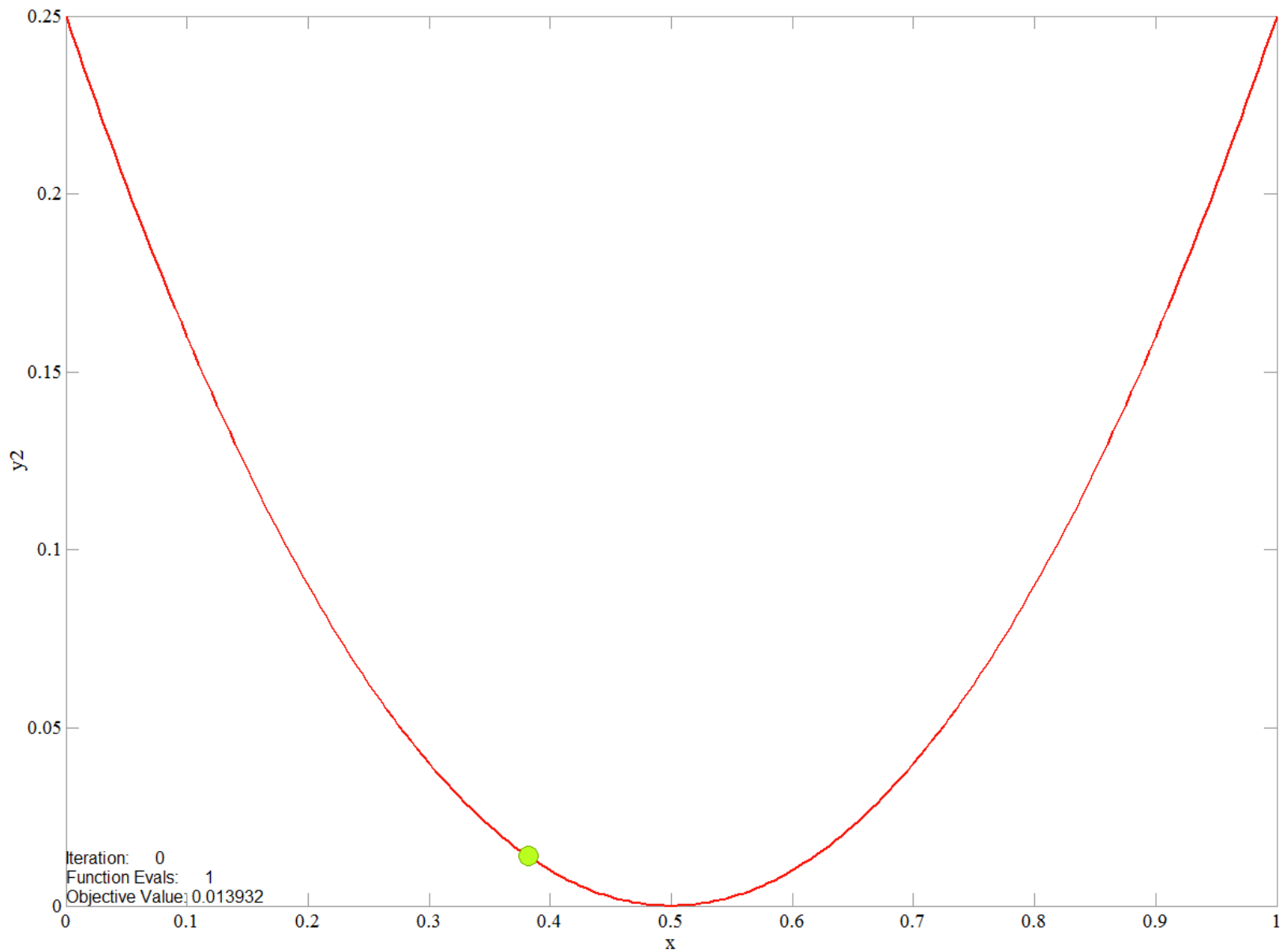
So, we can estimate the number of function calls needed to achieve a given tolerance...

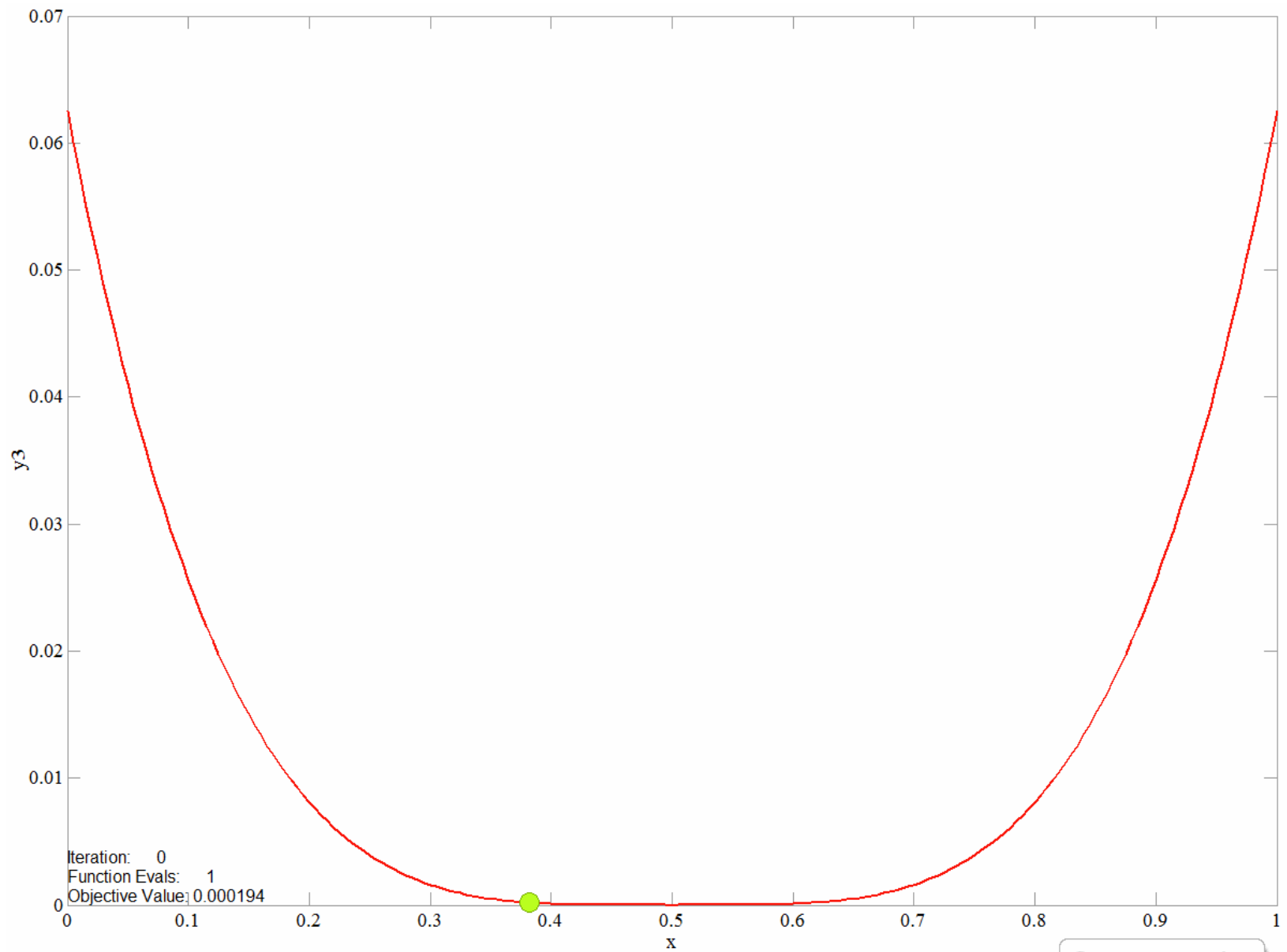
... or we can estimate the relative reduction in the bounds on the minimum for a specified “budget” of function calls.

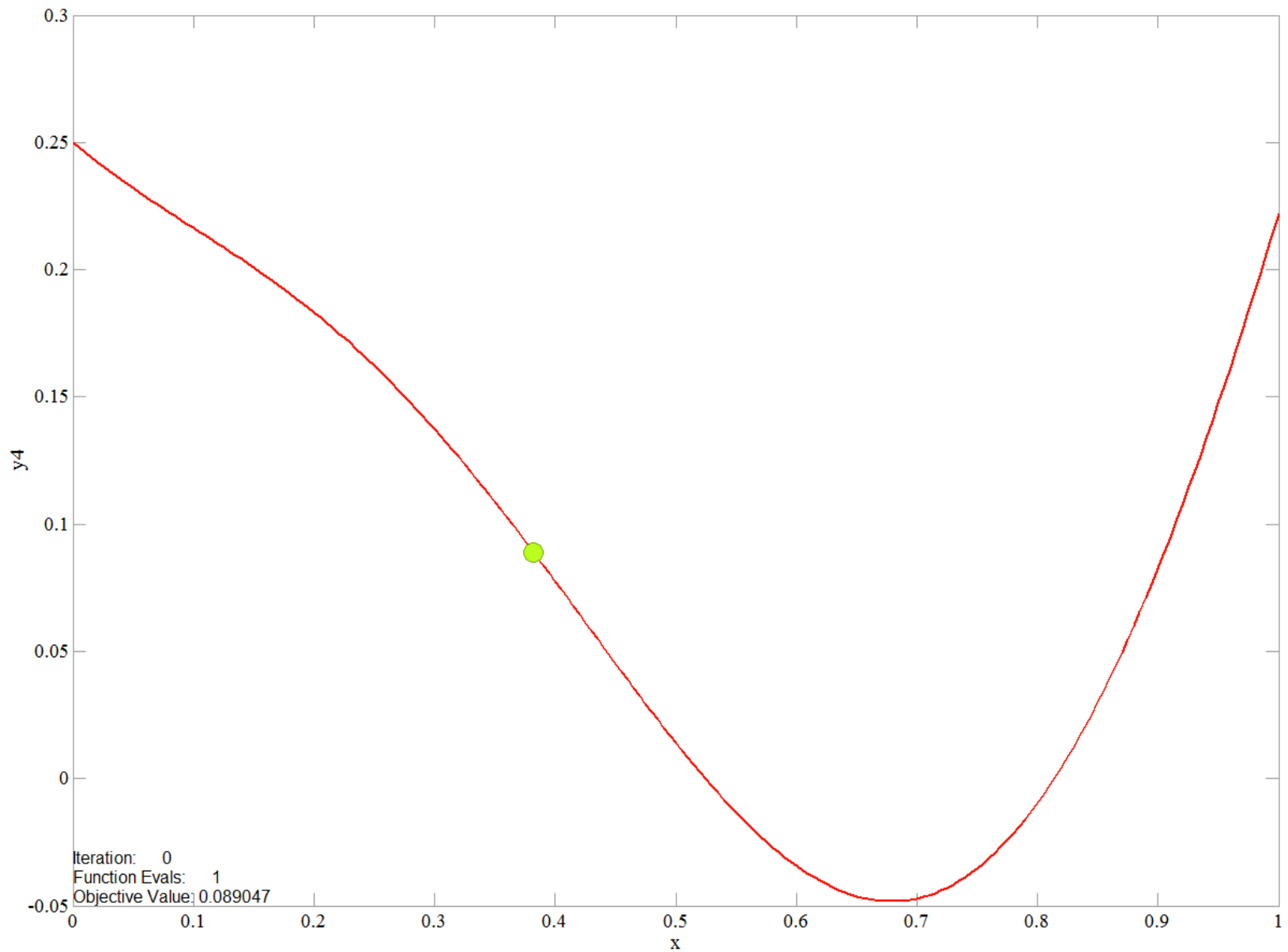
So let's look at some examples of the GSM method at work...





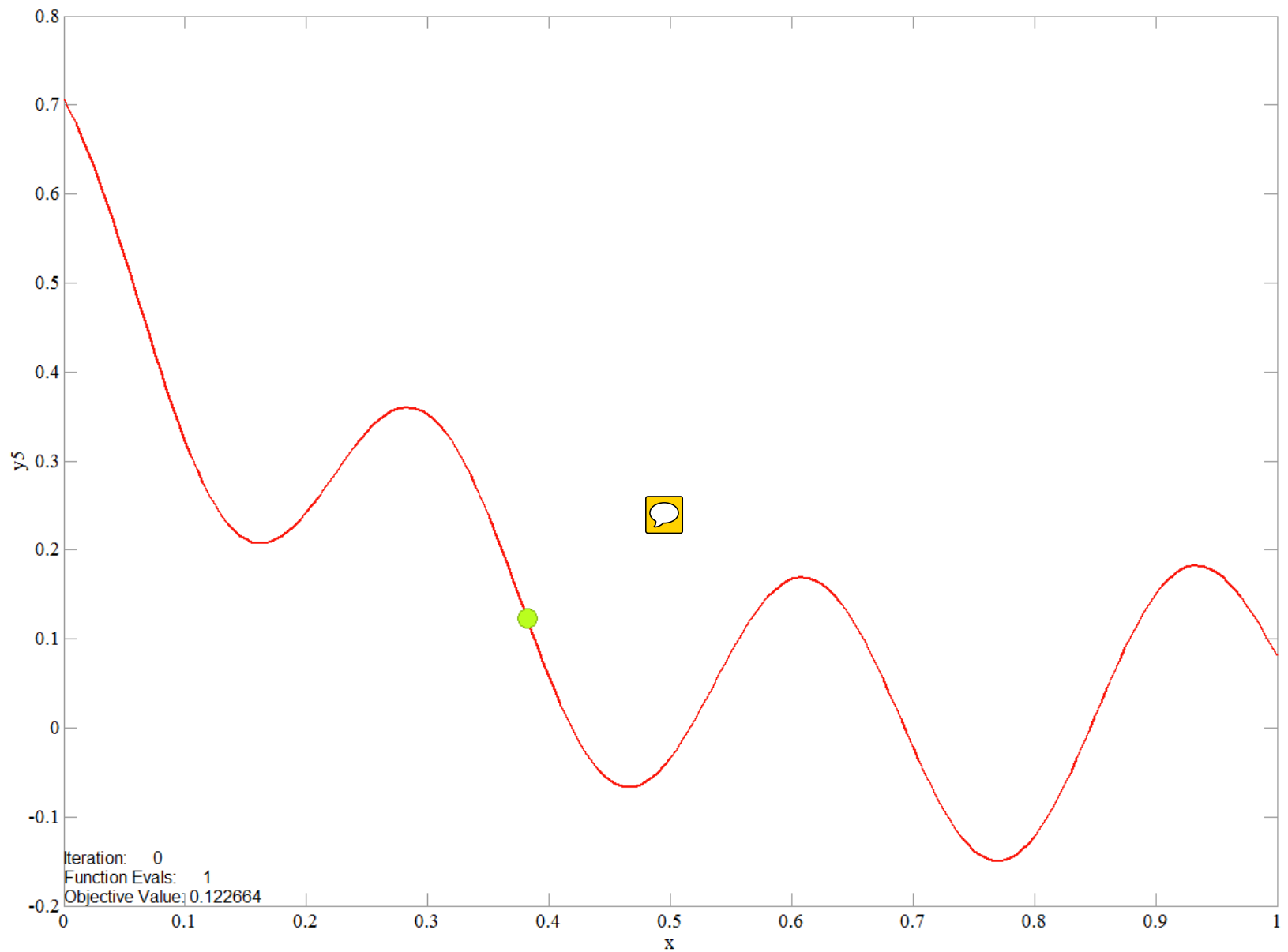






Iteration: 0
Function Evals: 1
Objective Value: 0.089047





When to Stop a Line Search – The Wolfe Conditions

The Wolfe conditions are specifications of “allowable” step lengths α in a line search.

Recall that seek an α^* that is “good enough” and not necessarily exact. The Wolfe conditions give quantitative meaning to “good enough.”

The Wolfe conditions could be applied in each line search and do not necessarily measure convergence of the overall algorithm.



Wolfe Conditions

There are two Wolfe conditions:

1. $f(\mathbf{x}_{k-1} + \alpha \mathbf{s}_k) \leq f(\mathbf{x}_{k-1}) + c_1 \alpha \mathbf{s}_k^T \nabla f(\mathbf{x}_{k-1})$
2. $\mathbf{s}_k^T \nabla f(\mathbf{x}_{k-1} + \alpha \mathbf{s}_k) \geq c_2 \mathbf{s}_k^T \nabla f(\mathbf{x}_{k-1})$

with $0 < c_1 < c_2 < 1$. Typically, $c_1 \sim 10^{-4}$ and $c_2 \sim 0.9$.

Condition 1 is called the **Armijo rule**.

Condition 2 is called the **curvature condition**.



Wolfe Conditions

Note that all “small enough” values of α satisfy the first Wolfe condition.

Enforcing only this condition could result in tediously small steps.

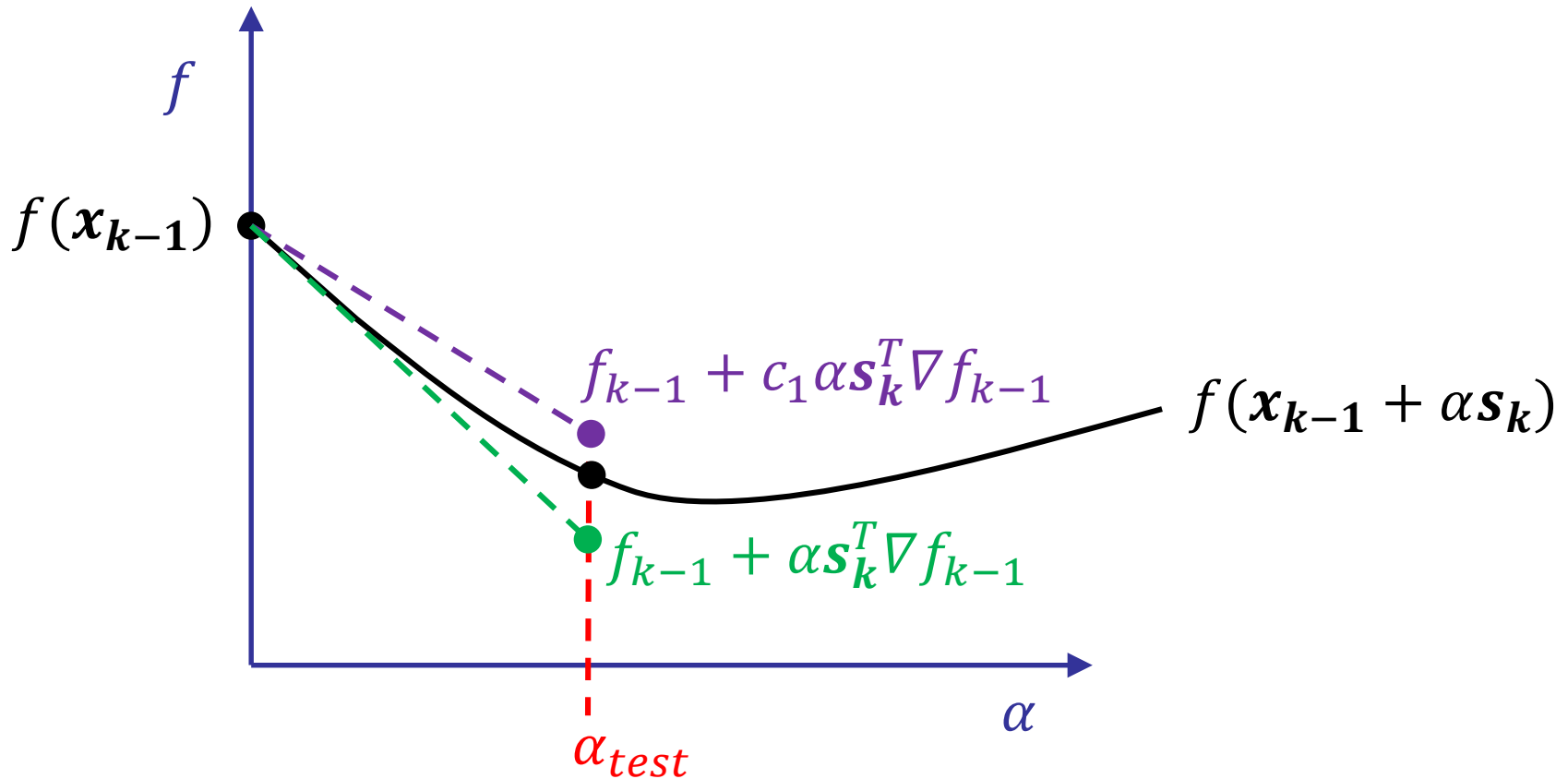
The curvature condition helps to prevent step sizes that are too small, i.e. it requires a sufficient increase in the slope of the function along the search line.

(Remember that the derivative starts off negative in the search direction, and we are trying to make it go to zero)



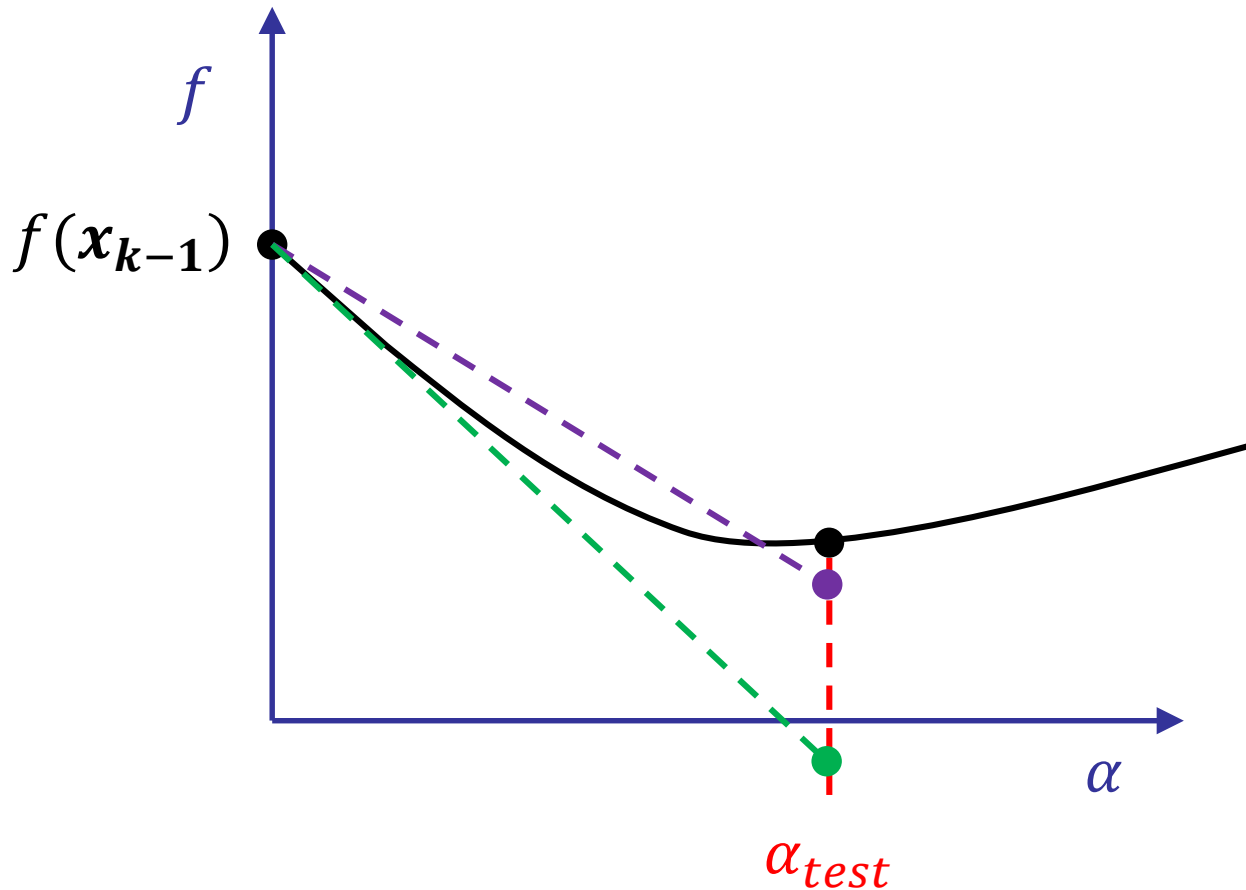
Wolfe Conditions

An α that satisfies the first Wolfe condition (Armijo rule):



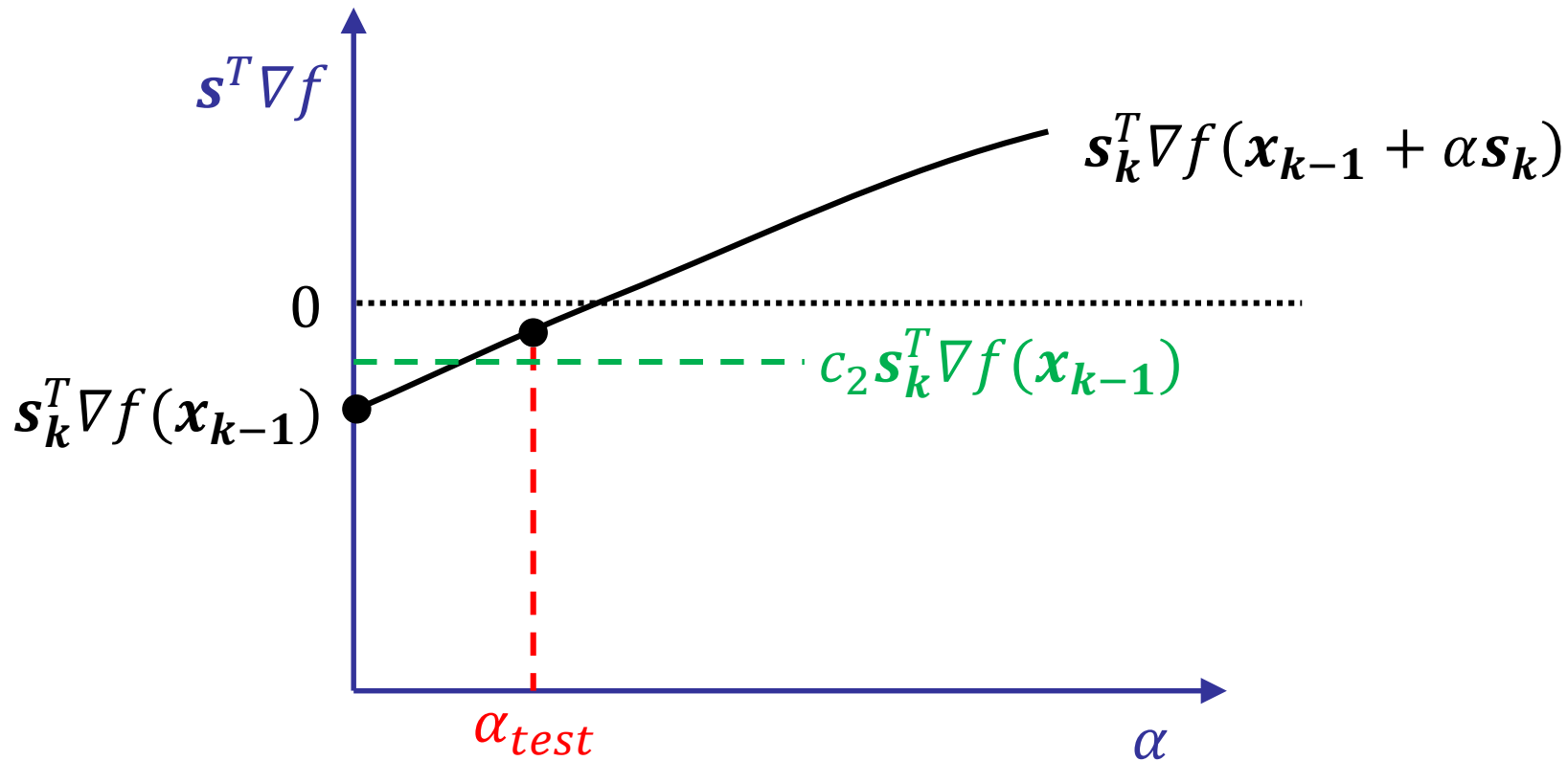
Wolfe Conditions

An α that does not satisfy the first Wolfe condition:



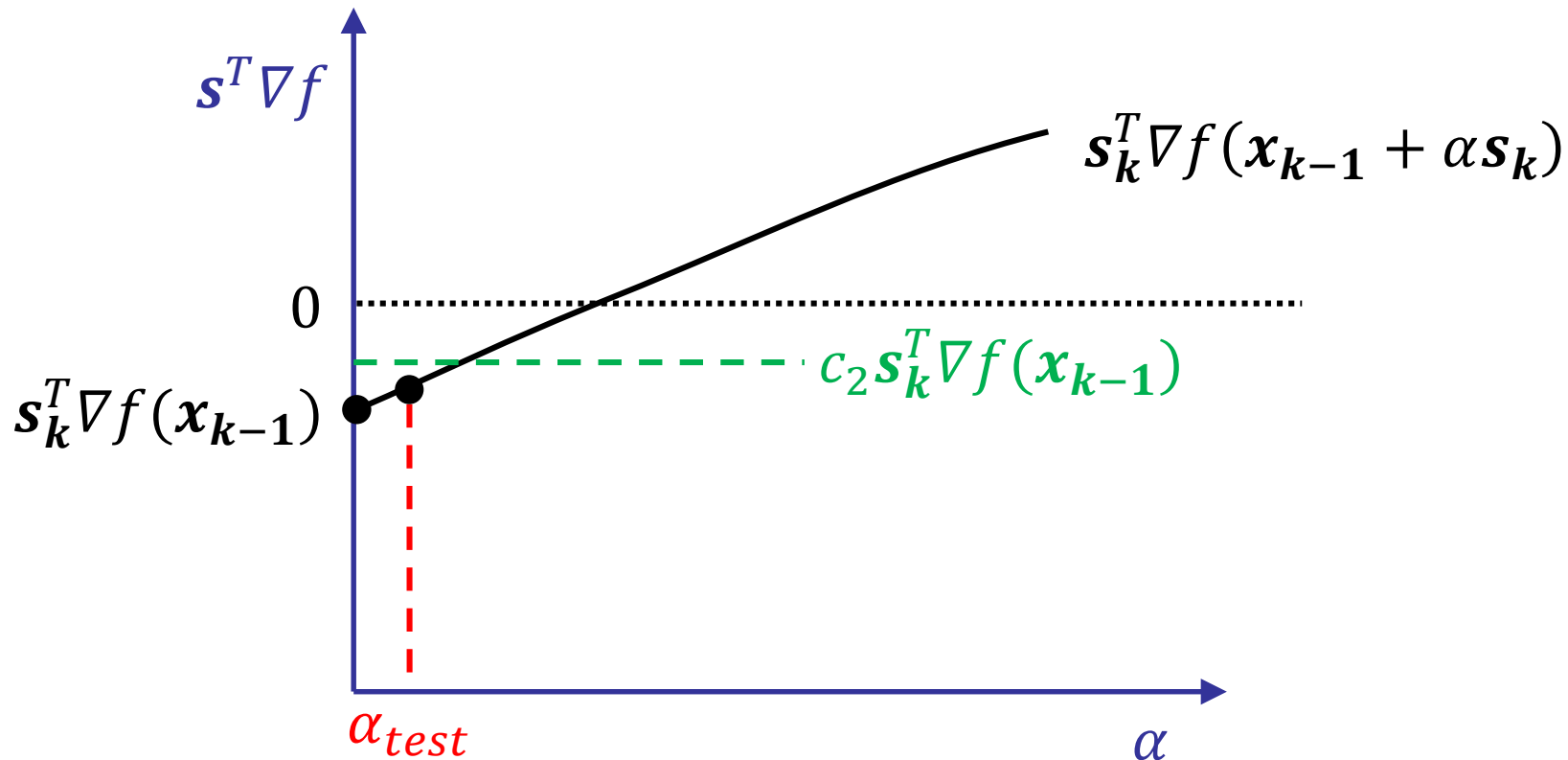
Wolfe Conditions

An α that satisfies the second Wolfe condition:



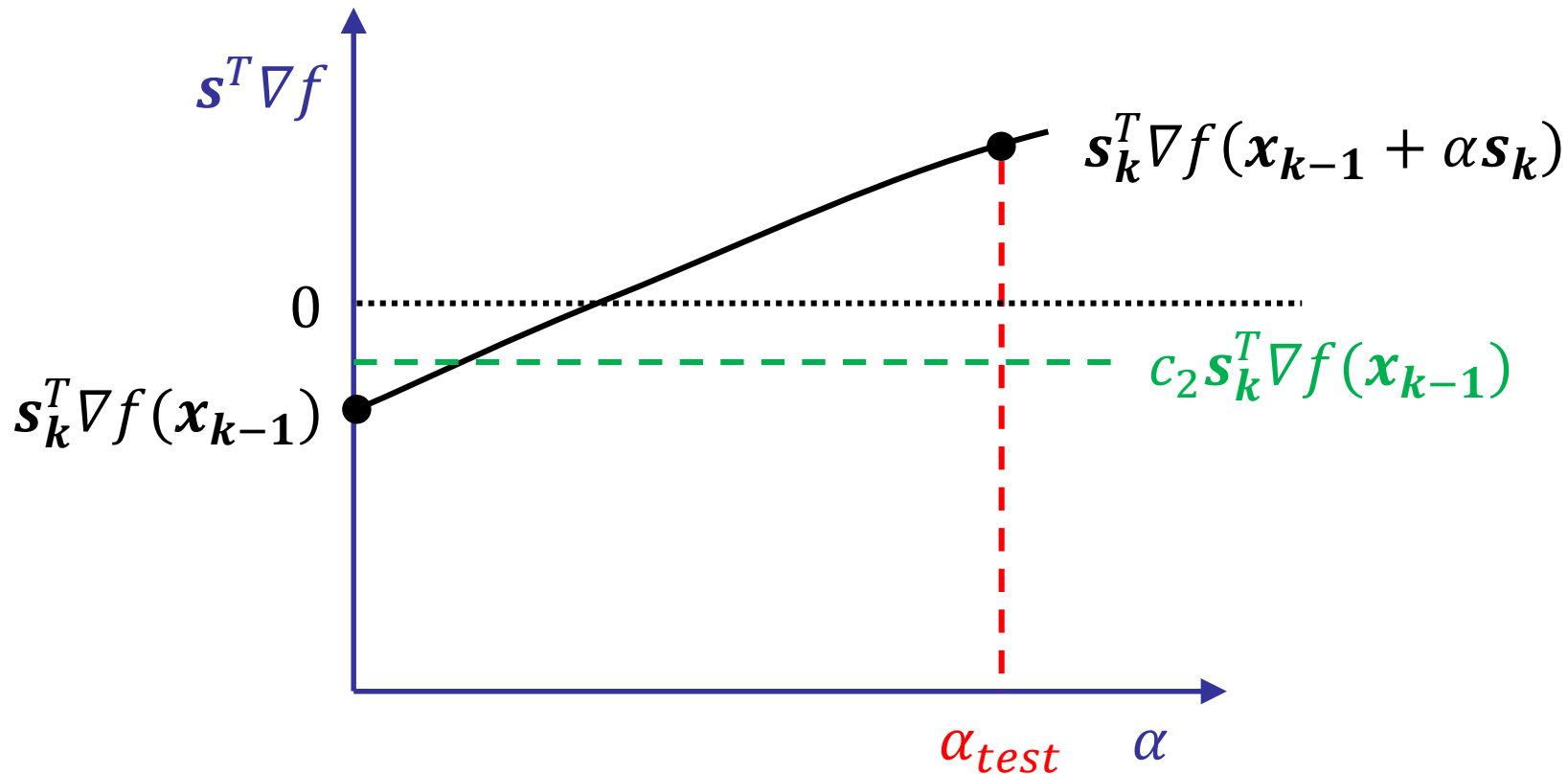
Wolfe Conditions

An α that does not satisfy the second Wolfe condition (curvature):



Wolfe Conditions

Another α that satisfies the second Wolfe condition:



Wolfe Conditions

The point α_{test} on the last slide meets the curvature condition, but it could be argued that it's not a very good approximation of the optimal α .

The issue is that the magnitude of the derivative in the search direction is rather large at this α_{test} .

This issue motivates the development of the ***strong*** Wolfe conditions.



Strong Wolfe Conditions

The strong Wolfe conditions are as follows

1. $f(\mathbf{x}_{k-1} + \alpha \mathbf{s}_k) \leq f(\mathbf{x}_{k-1}) + c_1 \alpha \mathbf{s}_k^T \nabla f_{k-1}$
2. $|\mathbf{s}_k^T \nabla f(\mathbf{x}_{k-1} + \alpha \mathbf{s}_k)| \leq c_2 |\mathbf{s}_k^T \nabla f(\mathbf{x}_{k-1})|$

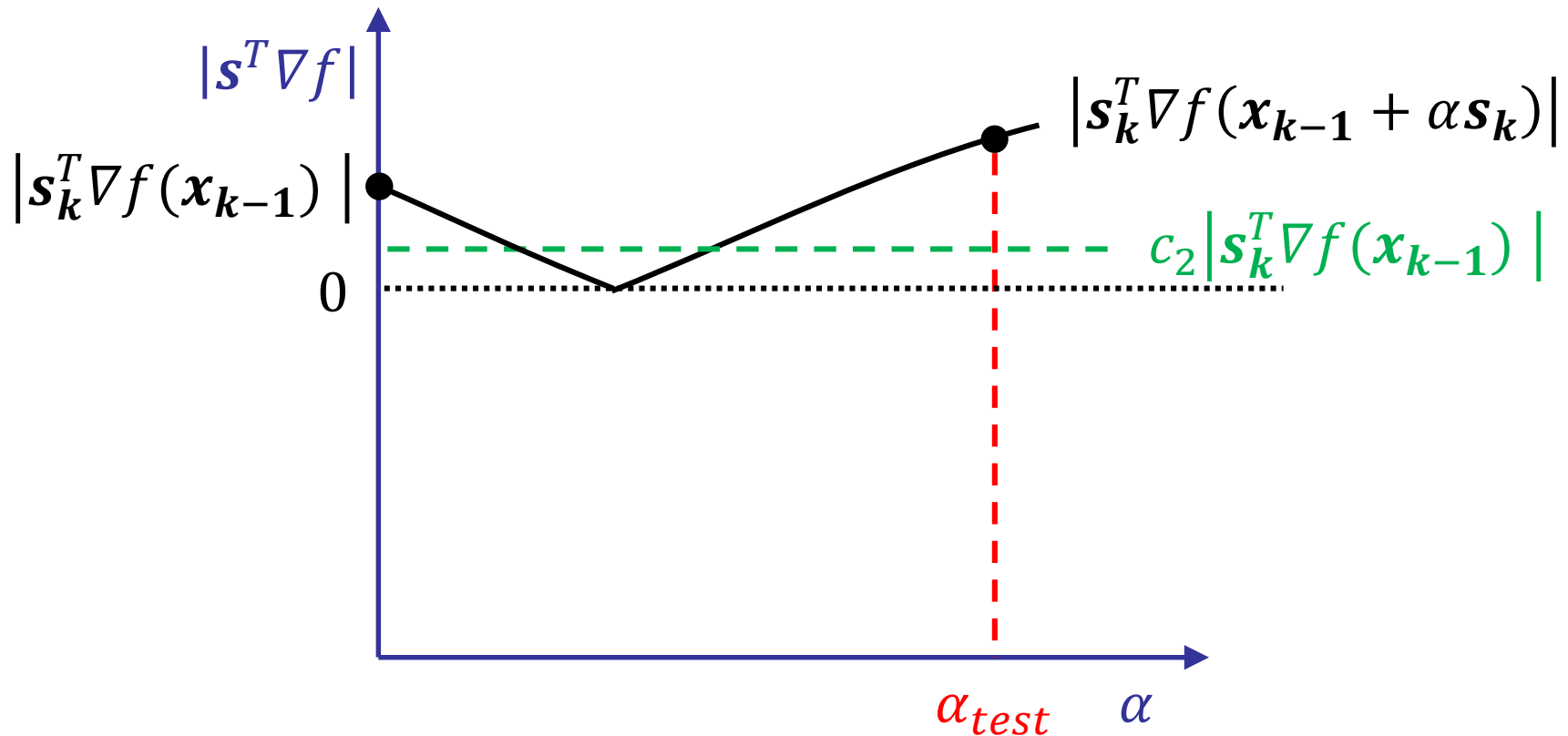
with $0 < c_1 < c_2 < 1$.

Only the curvature condition is changed compared to the original Wolfe conditions; specifically, it is modified to preclude values of α that increase the *magnitude* of the derivative.



Strong Wolfe Conditions

The α from slide 24 does not satisfy the strong curvature condition:



Backtracking and the Wolfe Conditions

If we begin a line search with a “large” α and then successively decrease it by a constant percentage, it can be shown that it is necessary to check only the first Wolfe condition (Armijo rule) and not the curvature condition.

This process of decreasing α by a constant percentage is called *backtracking*.

See Nocedal and Wright, Chapter 3, for details.

Also: Think about how the Wolfe conditions relate to bracketing a minimum...

