Fixed Point Finder

Lakshmi Krishnamurthy

v0.26, 14 August 2013

Framework Glossary

1. Hyperspace Search: Hyperspace search is a search to determine whether the entity is inside the zone of a range, e.g., bracketing search.
2. Hyperpoint Search: Hyperpoint searches are hard searches that search for an exact specific point (to within an appropriately established tolerance).
3. Iterate Nodes: This is the set of the traveled nodes (variate/Objective Function ordered pairs) that contain the trajectory traveled.
4. Iteration Search Primitives: The set of variate iteration routines that generate the subsequent iterate nodes.
5. Compound iterator search scheme: Search schemes where the primitive iteration routine to be invoked at each iteration are evaluated.
6. RunMap: Map that holds the program state at the end of each iteration, in the generic case, this is composed of the Wengert iterate node list, along with the corresponding root finder state.
7. Cost of Primitive (cop): This is the cost of invocation of a single variate iterator primitive.

Purpose of the Framework

1. Basic Search/Optimization SKU Framework
2. Unconstrained 1-D/multi-dimensional search
3. Constrained multi-dimensional Search
4. Pattern/String Search
5. Image Search

Framework

1. The root search given an objective function and its goal is achieved by iteratively evolving the variate, and involves the following steps:
    - Search initialization and root reachability determination: Searched is kicked off by spawning a root variate iterator for the search initialization process (described in detail in the next section).
    - Absolute Tolerance Determination.
    - Root Search Iteration: The root is searched iteratively according to the following steps:
        1. The iterator progressively reduces the bracket width.
        2. Successive iteration occurs using either a single primitive (e.g., using the bisection primitive), or using a selector scheme that picks the primitives for each step (e.g., Brent's method).
        3. For Open Method, instead of 1 and 2, the routine drives towards convergence iteratively.
    - Search Termination Detection: The search termination occurs typically based on the following:
        - Proximity to the Objective Function Goal
        - Convergence on the variate
        - Exhaustion if the number of iterations
2. The flow behind these steps is illustrated in Figure 1.
3. The "Flow Control Variate" in root search is the "Objective Function Distance to Goal" Metric.

Search Initialization

1. Broadly speaking, root finding approaches can be divided into a) those that bracket roots before they solve for them, and b) those that don't need to bracket, opting instead to pick a suitable starting point.

2. Depending upon the whether the search is a bracketing or an open method, the search initialization does one the following:
    - Determine the root brackets for bracketing methods
    - Locate root convergence zone for open methods
3. Initialization begins by a search for the starting zone. A suitable starting point/zone is determined where, by an appropriate choice for the iterator, you are expected to reach the fixed-point target within a sufficient degree of reliability. Very general-purpose heuristics often help determine the search start zone.
4. Both bracketing and open initializers are hyperspace searches, since they search for something “IN”, not “AT”.

Bracketing Start Initialization

1. Bracketing is the process of localizing the fixed point to within a target zone with the least required number of Objective Function calculations. Steps are:
    - Determine a valid bracketing search start
    - Choose a suitable bracket expansion
    - Limit attention to where the Objective Function is defined (more on this below).
2. Figure 2 shows the flow for the Bracketing routine.
3. Bracketing methods require that the initial search interval bracket the root (i.e. the function values at interval end points have opposite signs).
4. Bracketing traps the fixed point between two variate values, and uses the intermediate value theorem and the continuity of the Objective Function to guarantee the presence/existence of the fixed point between them.
5. Unless the objective function is discontinuous, bracketing methods guarantee convergence (although may not be within the specified iteration limit).
6. Typically, they do not require the objective function to be differentiable.
7. Bracketing iteration primitives’ convergence is usually linear to super-linear.
8. Bracketing methods preserve bracketing throughout computation and allow user to specify which side of the convergence interval to select as the root.

9. It is also possible to force a side selection after a root has been found, for example, in sequential search, to find the next root.
10. Generic root bracketing methods that treat the objective function as a black box will be slower than targetted ones – so much so that they can constitute the bulk of the time for root search. This is because, to accommodate generic robustness coupled with root-pathology avoidance (oscillating bracket pairs etc), these methods have to perform a full variate space sweep without any assumptions regarding the location of the roots (despite this most bracketing algorithms cannot guarantee isolation of root intervals). For instance, naïve examination of the Objective Function's "sign-flips" alone can be misleading, especially if you bracket fixed-points with even numbered multiplicity within the brackets. Thus, some ways of analyzing the Black Box functions (or even the closed form Objective Functions) are needed to better target/customize the bracketing search (of course, parsimony in invoking the number of objective function calls is the main limitation).
11. The first step is to determine a valid bracketing search start. One advantage with univariate root finding is that objective function range validity maybe established using an exhaustive variate scanner search without worrying about combinatorial explosion.

Objective Function Failure

1. Objective Function may fail evaluation at the specified variate for the following reason:

- Objective Function is not defined at the specified variate.
- Objective Function evaluates to a complex number.
- Objective Function evaluation produces NaN/Infinity/Under-flow/Over-flow errors.
- In such situations, the following steps are used to steer the variate to a valid zone.

2. Objective Function undefined at the Bracketing Candidate Variate: If the Objective Function is undefined at the starting variate, the starting variate is expanded using the variate space scanner algorithm described above. If the objective Function like what is seen in Figure 3, a valid starting variate will eventually be encountered.
3. Objective Function not defined at any of the Candidate Variates: The risk is that the situation in Figure 4 may be encountered, where the variate space scanner iterator "jumps over" the range over which the objective function is defined. This could be because the objective function may have become complex. In this case, remember that an even power of the objective function also has the same roots as the objective function itself. Thus, solving for an even power of the objective function (like the square) – or even bracketing for it – may help.

Bracketing Start Initialization

1. Figure 5 shows the flow behind a general-purpose bracket start locator.
2. Once the starting variate search is successful, and the objective function validity is range-bound, then use an algorithm like bisection to bracket the root (as shown in Figure 6 below).
3. However, if the objective function runs out of its validity range under the variate scanner scheme, the following steps need to undertaken:
   - If the left bracketing candidate fails, bracketing is done towards the right using the last known working left-most bracketing candidate as the "left edge".
   - Likewise, if the right bracketing candidate fails, bracketing is done towards the left using the last known working right-most bracketing candidate as the "right edge".
4. The final step is to trim the variate zone. Using the variate space scanner algorithm, and the mapped variate/Objective Function evaluations, the tightest bracketing zones are extracted (Figure 7).

Open Search Initialization

1. Non-bracketing methods use a suitable starting point to kick off the root search. As is obvious, the chosen starting point can be critical in determining the fate of the search. In particular, it should be within the zone of convergence of the fixed-point root to guarantee convergence. This means that specialized methods are necessary to determine zone of convergence.
2. When the objective function is differentiable, the non-bracketing root finder often may make use of that to achieve quadratic or higher speed of convergence. If the non-bracketing root finder cannot/does not use the objective function's differentiability, convergence ends up being linear to super-linear.
3. The typical steps for determining the open method starting variate are:
   - Find a variate that is proximal to the fixed point
   - Verify that it satisfies the convergence heuristic
4. Bracketing followed by a choice of an appropriate primitive variate (such as bisection/secant) satisfies both, thus could be a good starting point for open method searches like Newton's method.
5. Depending upon the structure of the Objective Function, in certain cases the chain rule can be invoked to ease the construction of the derivative – esp. in situations where the sensitivity to inputs are too high/low.

Search/Bracketing Initializer Heuristic Customization

1. Specific Bracketing Control Parameters
2. Left/Right Soft Bracketing Start Hints: The other components may be used from the bracketing control parameters.
3. Mid Soft Bracketing Start Hint: The other components may be used from the bracketing control parameters.
4. Floor/Ceiling Hard Bracketing Edges: The other components may be used from the bracketing control parameters.

5. Left/Right Hard Search Boundaries: In this case, no bracketing is done – brackets are used to verify the roots, search then starts directly.

Numerical Challenges in Root Finding

1. Bit Cancellation
2. Ill-conditioning (e.g., see high order polynomial roots)
3. "domains of indeterminacy" – existence of sizeable intervals around which the objective function hovers near the target
4. Continuous, but abrupt changes (e.g., near-delta Gaussian objection function)
5. Under-flow/over-flow/roundoff errors
6. root multiplicity (e.g., in the context of polynomial roots)
7. Typical solution is to transform the objective function to a better conditioned function – insight into the behavior of the objective can be used to devise targetted solutions.

Variate Iteration

1. $\nu_{i+1} = I(\nu_i, \Im_i)$ where ν_i is the ith variate and $\Im_i$ is the root finder state after the ith iteration.
2. Iterate nodes as Wengert variables: Unrolling the traveled iterate nodes during the forward accumulation process, as a Wengert list, is a proxy to the execution time, and may assist in targeted pre-accumulation and check-pointing.
3. Cognition Techniques of Mathematical Functions:
    - Wengert Variate Analysis => Collection of the Wengert variates helps build and consolidate the Objective Function behavior from the variate iterate Wengert nodes – to build a behavioral picture of the Objective Function.
    - Objective Function Neighborhood Behavior => With every Wengert variable, calculation of the set of forward sensitivities and the reverse Jacobians builds a local picture of the Objective Function without having to evaluate it.

4. Check pointing: Currently implemented using a roving variate/OF iterate node "RunMap"; this is also used to check circularity in the iteration process.
5. Compound Iterator RunMap: For compound iterations, the iteration circularity is determined the doublet $\{\nu_i, \Im_i\}$, so the Wengert RunMap is really a doublet Multi-Map.
6. Hyperpoint univariate fixed point search proximity criterion: For hyperpoint checks, the search termination check needs to explicitly accommodate a "proximity to target" metric. This may not be then case for hyperspace checks.
7. Regime crossover indicator: On one side the crossover, the variate is within the fast convergence zone, so you may use faster Open techniques like the Newton's methods. On the other side, continue using the bracketing techniques.
    a. Fast side of the crossover must be customizable (including other Halley's method variants); robust side should also be customizable (say False Position).
8. Crossover indicator determination: Need to develop targeted heuristics needed to determine the crossover indicator.
    - Entity that determines the crossover indicator may be determined from the relative variate shift change $\frac{x_{N+1} - x_N}{x_N - x_{N-1}}$ and the relative objective function change $\frac{y_{N+1} - y_N}{y_N - y_{N-1}}$.
9. Types of bracketing primitives:
    - Bracket narrower primitives (Bisection, false position), and interpolator primitives (Quadratic, Ridder).
    - Primitive's COP determinants: Expressed in terms of characteristic compute units.
        a. Number of objective function evaluation (generally expensive).
        b. Number of variate iterator steps needed.
        c. Number of objective function invocation per a given variate iteration step.
    - Bracket narrower primitives => Un-informed iteration primitives, low invocation cost (usually single objective function evaluation), but low search targeting quality, and high COP.

- Interpolator primitives => Informed iteration primitives, higher invocation cost (multiple objective function evaluations, usually 2), better search targeting quality, and lower COP.

10. Pre-OF Evaluation Compound Heuristic: Heuristic compound variates are less informed, but rely heavily on heuristics to extract the subsequent iterator, i.e., pre-OF evaluation heuristics try to guide the evolution without invoking the expensive OF evaluations (e.g., Brent, Zheng).
11. OF Evaluation Compound Heuristic: These compound heuristics use the OF evaluations as part of the heuristics algorithm to establish the next variate => better informed

Open Root Search Method: Newton's method

1. Newton's method uses the objective function *f* and it's derivative *f*' to iteratively evaluate the root.
2. Given a well-behaved function *f* and it's derivative *f*' defined over real *x*, the algorithm starts with an initial guess of x_0 for the root.
3. First iteration yields $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$.
4. This is repeated in $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ till a value x_n that is convergent enough is obtained.
5. If α is a simple root (root with multiplicity 1), and $\varepsilon_n = x_n - \alpha$ and $\varepsilon_{n+1} = x_{n+1} - \alpha$ respectively, then for sufficiently large *n*, the convergence is quadratic:

$$\varepsilon_{n+1} \approx \frac{1}{2}\left|\frac{f''(x_n)}{f'(x_n)}\right|\varepsilon_n^2$$

6. Newton's method only works when *f* has continuous derivatives in the root neighborhood.
7. When analytical derivatives are hard to compute, calculate slope through nearby points, but convergence tends to be linear (like secant).

8. If the first derivative is not well behaved/does not exit/undefined in the neighborhood of a particular root, the method may overshoot, and diverge from that root.
9. If a stationary point of the function is encountered, the derivative is zero and the method will fail due to division by zero.
10. The stationary point can be encountered at the initial or any of the other iterative points.
11. Even if the derivative is small but not zero, the next iteration will be a far worse approximation.
12. A large error in the initial estimate can contribute to non-convergence of the algorithm (owing to the fact that the zone is outside of the neighborhood convergence zone).
13. If α is a root with multiplicity $m > 1$, then for sufficiently large *n*, the convergence becomes linear - $\varepsilon_{n+1} \approx \frac{m-1}{m}\varepsilon_n$
14. When there are two or more roots that are close together then it may take many iterations before the iterates get close enough to one of them for the quadratic convergence to be apparent.
15. However, if the multiplicity *m* of the root is known, one can use the following modified algorithm that preserves the quadratic convergence rate (equivalent to using successive over-relaxation)

$$x_{n+1} = x_n - m\frac{f(x_n)}{f'(x_n)}$$

16. The algorithm estimates *m* after carrying out one or two iterations, and then use that value to increase the rate of convergence. Alternatively, the modified Newton's method may also be used:

$$x_{n+1} = x_n - \frac{f(x_n)f'(x_n)}{\left[f'(x_n)f'(x_n) - f(x_n)f''(x_n)\right]}$$

17. It is easy to show that if $f'(x_N) = 0$ and $f''(x_N) \neq 0$, the convergence in the neighborhood becomes linear. Further, if $f'(x_N) \neq 0$ and $f''(x_N) = 0$, convergence becomes cubic.
18. One way of determining the neighborhood of the root. Define

$$g(x) = x - \frac{f(x)}{f'(x)}$$

$$p_n = g(p_{n-1})$$

where

(a) p is a fixed point of g [$g, g' \in C[a,b]$]

(b) k is a positive constant,

(c) $p_0 \in C[a,b]$, and

(e) $g(x) \in C[a,b]$ for all $x \in [a,b]$.

19. One sufficient condition for p_0 to initialize a convergent sequence $\{p_k\}_{k=0}^{\infty}$, which converges to the root $x = p$ of $f(x) = 0$ is that $(p-\delta, p+\delta)$ and that δ be chosen so that $\frac{f(x)f''(x)}{f'(x)f'(x)} \le k < 1$ for all $x \in (p-\delta, p+\delta)$.

20. It is easy to show that under specific choices for the starting variate, Newton's method can fall into a basin of attraction. These are segments of the real number line such that within each region iteration from any point leads to one particular root - can be infinite in number and arbitrarily small. Also, the starting or the intermediate point can enter a cycle - the n-cycle can be stable, or the behavior of the sequence can be very complex (forming a Newton fractal).
21. Newton's method for optimization is equivalent to iteratively maximizing a local quadratic approximation to the objective function. But some functions are not approximated well by quadratic, leading to slow convergence, and some have turning points where the curvature changes sign, leading to failure. Approaches to fix this use a more appropriate choice of local approximation than quadratic, based on the type of function we are optimizing. [13] demonstrates three such generalized Newton rules. Like Newton's method, they only involve the first two derivatives of the function, yet converge faster and fail less often.

22. One significant advantage of Newton's method is that it can be readily generalized to higher dimensions.
23. Also, Newton's method calculates the Jacobian automatically as part of the calibration process, owing to the reliance on derivatives – in particular, automatic differentiation techniques can be effectively put to use.

Secant

1. Secant method results on the replacement of the derivative in the Newton's method with a secant-based finite difference slope.
2. Convergence for the secant method is slower than the Newton's method (approx. order is 1.6); however, the secant method does not require the objective function to be explicitly differentiable.
3. It also tends to be less robust than the popular bracketing methods.

Bracketing Iterative Root Search

1. Bracketing iterative root searches attempt to progressively narrow the brackets and to discover the root within.
2. The first set discusses the goal search univariate iterator primitives that are commonly used to iterate through the variate.
3. These goal search iterator primitives continue generating a new pair of iteration nodes (just like their bracketing search initialization counter-parts).
4. Certain iterator primitives carry bigger "local" cost, i.e., cost inside a single iteration, but may reduce global cost, e.g., by reducing the number iterations due to faster convergence.
5. Further, certain primitives tend to be inherently more robust, i.e., given enough iteration, they will find the root within – although they may not be fast.

6. Finally the case of compound iterator search schemes, search schemes where the primitive iteration routine to be invoked at each iteration is evaluated on-the-fly, are discussed.
7. Iterative searches that maintain extended state across searches pay a price in terms of scalability – the price depending on the nature and the amount of state held (e.g., Brent's method carries iteration selection state, whereas Zheng's does not).

Univariate iterator primitive: Bisection

1. Bisection starts by determining a pair of root brackets *a* and *b*.
2. It iteratively calculates f at $c = \frac{a+b}{2}$, then uses *c* to replace either *a* or *b*, depending on the sign. It eventually stops when *f* has attained the desired tolerance.
3. Bisection relies on *f* being continuous within the brackets.
4. While the method is simple to implement and reliable (it is a fall-back for less reliable ones), the convergence is slow, producing a single bit of accuracy with each iteration.

Univariate iterator primitive: False Position

1. False position works the same as bisection, except that the evaluation point *c* is linearly interpolated; *f* is computed at $c = \frac{af(b)+bf(a)}{f(b)+f(a)}$, where *f(a)* and *f(b)* have opposite signs. This holds obvious similarities with the secant method.
2. False position method also requires that *f* be continuous within the brackets.
3. It is simple enough, more robust than secant and faster than bisection, but convergence is still linear to super-linear.
4. Given that the linear interpolation of the false position method is a first-degree approximation of the objective function within the brackets, quadratic approximation using Lagrange interpolation may be attempted as

$$f(x)=\frac{(x-x_{n-1})(x-x_n)}{(x_{n-2}-x_{n-1})(x_{n-2}-x_n)}f_{n-2}+\frac{(x-x_{n-2})(x-x_n)}{(x_{n-1}-x_{n-2})(x_{n-1}-x_n)}f_{n-1}+\frac{(x-x_{n-2})(x-x_{n-1})}{(x_n-x_{n-2})(x_n-x_{n-1})}f_n$$

where we use the three iterates, x_{n-2}, x_{n-1} and x_n, with their function values, f_{n-2}, f_{n-1} and f_n.

5. This reduces the number of iterations at the expense of the function point calculations.
6. Using higher order polynomial fit for the objective function inside the bracket does not always produce roots faster or better, since it may result in spurious inflections (e.g., Runge's phenomenan).
7. Further, quadratic or higher fits may also cause complex roots.

Univariate iterator primitive: Inverse Quadratic

1. Performing a fit of the inverse *1/f* instead of *f* avoids the quadratic interpolation problem above. Using the same symbols as above, the inverse can be computed as

$$f^{-1}(y)=\frac{(y-f_{n-1})(y-f_n)}{(f_{n-2}-f_{n-1})(f_{n-2}-f_n)}f_{n-2}+\frac{(y-f_{n-2})(y-f_n)}{(f_{n-1}-f_{n-2})(f_{n-1}-f_n)}f_{n-1}+\frac{(y-f_{n-2})(y-f_{n-1})}{(f_n-f_{n-2})(f_n-f_{n-1})}x_n$$

2. Convergence is faster than secant, but poor when iterates not close to the root, e.g., if two of the function values f_{n-2}, f_{n-1} and f_n coincide, the algorithm fails.

Univariate iterator primitive: Ridder's

1. Ridders' method is a variant on the false position method that uses exponential function to successively approximate a root of *f*.
2. Given the bracketing variates, x_1 and x_2, which are on two different sides of the root being sought, the method evaluates f at $x_3=\frac{x_1+x_2}{2}$.

3. It extracts exponential factor α such that $f(x)e^{\alpha x}$ forms a straight line across $x_{1,}x_{2,}$ and x_3. x_2 is calculated from

$$x_4 = x_3 + (x_3 - x_1)\frac{sign[f(x_1) - f(x_2)]f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}$$

2. Ridder's method is simpler than Brent's method, and has been claimed to perform about the same.
3. However, the presence of the square root can render it unstable for many of the reasons discussed above.

Univariate compound iterator: Brent and Zheng

1. Brent's predecessor method first combined bisection, secant, and inverse quadratic to produce the optimal root search for the next iteration.
2. Starting with the bracket points a_0 and b_0, two provisional values for the next iterate are computed; the first given by the secant method $s = b_k - \frac{b_k - b_{k-1}}{f(b_k) - f(b_{k-1})} f(b_k)$, and the second by bisection $m = \frac{a_k + b_k}{2}$.
3. If s lies between b_k and m, it becomes the next iterate b_{k+1}, otherwise the m is the next iterate.
4. Then, the value of the new contrapoint is chosen such that $f(a_{k+1})$ and $f(b_{k+1})$ have opposite signs.
5. Finally, if $|f(a_{k+1})| < |f(b_{k+1})|$, then a_{k+1} is probably a better guess for the solution than b_{k+1}, and hence the values of a_{k+1} and b_{k+1} are exchanged.
6. To improve convergence, Brent's method requires that two inequalities must be simultaneously satisfied.
   a) Given a specific numerical tolerance δ, if the previous step used the bisection method, if $\delta < |b_k - b_{k-1}|$, the bisection method is performed and its result used for

the next iteration. If the previous step used interpolation, then the check becomes $\delta < |b_{k-1} - b_{k-2}|$.

b) If the previous step used bisection, if $|s - b_k| < \frac{1}{2}|b_k - b_{k-1}|$, secant is used; otherwise the bisection used for the next iteration. If the previous step performed interpolation, $|s - b_k| < \frac{1}{2}|b_{k-1} - b_{k-2}|$ is checked instead.

7. Finally, since Brent's method uses inverse quadratic interpolation, *s* has to lie between $(3a_k + b_k) / 4$ and b_k.
8. Brent's algorithm uses three points for the next inverse quadratic interpolation, or secant rule, based upon the criterion specified above.
9. One simplification to the Brent's method adds one more evaluation for the function at the middle point before the interpolation.
10. This simplification reduces the times for the conditional evaluation and reduces the interval of convergence.
11. Convergence is better than Brent's, and as fast and simple as Ridder's.

Polynomial Roots

1. This section carries out a brief treatment of computing roots for polynomials.
2. While closed form solutions are available for polynomials up to degree 4, they may not be stable numerically.
3. Popular techniques such as Sturm's theorem and Descartes' rule of signs are used for locating and separating real roots.
4. Modern methods such as VCA and the more powerful VAS use these with Bisection/Newton methods – these methods are used in Maple/Mathematica.
5. Since the eigenvalues of the companion matrix to a polynomial correspond to the polynomial's roots, common fast/robust methods used to find them may also be used.

6. A number of caveats apply specifically to polynomial root searches, e.g., Wilkinson's polynomial shows why high precision is needed when computing the roots – proximal/other ill-conditioned behavior may occur.
7. Finally, special ways exist to identify/extract multiplicity in polynomial roots – they use the fact that *f(x)* and *f'(x)* share the root, and by figuring out their GCD.

Software Framework Component - Execution Initialization

1. ***ExecutionInitializer*** implements the initialization execution and customization functionality. It performs two types of variate initialization:
    - Bracketing initialization: This brackets the fixed point using the bracketing algorithm described above. If successful, a pair of variate/Objective Function coordinate nodes that bracket the root is generated. These brackets are eventually used by routines that iteratively determine the root. Bracketing initialization is controlled by the parameters in ***BracketingControlParams***.
    - Convergence Zone initialization: This generates a variate that lies within the convergence zone for the iterative determination of the fixed point using the Newton's method. Convergence Zone Determination is controlled by the parameters in ***ConvergenceControlParams***.
2. ***ExecutionInitializationOutput*** holds the output of the root initializer calculation. It contains the following fields:

Whether the initialization completed successfully the number of iterations, the number of objective function calculations, and the time taken for the initialization
The starting variate from the initialization

Software Framework Component - Bracketing

1. ***BracketingControlParams*** implements the control parameters for bracketing solutions. It provides the following parameters:

- The starting variate from which the search for bracketing begins
- The initial width for the brackets
- The factor by which the width expands with each iterative search
- The number of such iterations.

2. ***BracketingOutput*** carries the results of the bracketing initialization. In addition to the fields of ***ExecutionInitializationOutput***, ***BracketingOutput*** holds the left/right bracket variates and the corresponding values for the objective function.

Software Framework Component - Convergence

1. ***ConvergenceControlParams*** holds the fields needed for the controlling the execution of Newton's method. It does that using the following parameters:
   - The determinant limit below which the convergence zone is deemed to have been reached.
   - Starting variate from where the convergence search is kicked off.
   - The factor by which the variate expands across each iterative search.
   - The number of search iterations.
2. ***ConvergenceOutput*** extends the ***ExecutionInitializationOutput*** by retaining the starting variate that results from the convergence zone search.
   - ***ConvergenceOutput*** does not add any new field to ***ExecutionInitializationOutput***.

Software Framework Component - Differentiation

1. ***ObjectiveFunction*** provides the evaluation of the objective function and its derivatives for a specified variate.
   - Default implementation of the derivatives is meant for non-analytical black box objective functions.

2. ***DerivativeControl*** provides bumps needed for numerically approximating derivatives.
3. ***Differential*** holds the incremental differentials for the variate and the objective function.

Software Framework Component - Execution Customization

1. ***ExecutionControl*** implements the core root search execution control and customization functionality.
    a. It is used for a) calculating the absolute tolerance, and b) determining whether the ***ObjectiveFunction*** has reached the goal.
    b. ***ExecutionControl*** determines the execution termination using its ***ExecutionControlParams*** instance.
2. ***ExecutionControlParams*** holds the parameters needed for controlling the execution of the root finder.
    - ***ExecutionControlParams*** fields control the root search in one of the following ways:
        a. Number of iterations after which the search is deemed to have failed
        b. Relative ***ObjectiveFunction*** Tolerance Factor which, when reached by the objective function, will indicate that the fixed point has been reached
        c. Absolute Tolerance fall-back, which is used to determine that the fixed point has been reached when the relative tolerance factor becomes zero

Software Framework Component - Fixed Point Search

1. ***FixedPointFinder*** is the base abstract class that is implemented by customized invocations, e.g., Newton's method, or any of the bracketing methodologies.
    - It invokes the core routine for determining the fixed point from the goal.
    - ***ExecutionControl*** determines the execution termination.
2. ***FixedPointFinder*** main flow comprises of the following steps:

- Initialize the root search zone by determining either a) the brackets, or b) the starting variate.
- Compute the absolute ***ObjectiveFunction*** tolerance that establishes the attainment of the fixed point.
- Launch the variate iterator that iterates the variate.
- Iterate until the desired tolerance has been attained
- Return the root finder output.

3. Root finders that derive from this provide implementations for the following:
   - Variate initialization: They may choose either bracketing initializer, or the convergence initializer - functionality is provided for both in this module.
   - Variate Iteration: Variates are iterated using a) any of the standard primitive built-in variate iterators (or custom ones), or b) a variate selector scheme for each iteration.
4. ***FixedPointFinderOutput*** holds the result of the root search.
   - It contains the following fields:
     - Whether the search completed successfully
     - The number of iterations, the number of objective function base/derivative calculations, and the time taken for the search
     - The output from initialization
5. ***FixedPointFinderNewton*** customizes the ***FixedPointFinder*** for Open (Newton's) root finder functionality.
   - ***FixedPointFinderNewton*** applies the following customization:
     - Initializes the fixed point finder by computing a starting variate in the convergence zone
     - Iterating the next search variate using the Newton's method.
6. ***FixedPointFinderBracketing*** customizes the ***FixedPointFinder*** for bracketing based root finder functionality.
   - ***FixedPointFinderBracketing*** applies the following customization:
     - Initializes the root finder by computing the starting brackets
     - Iterating the next search variate using one of the specified variate iterator primitives.

- ***FixedPointFinderBracketing*** does not do compound iterations of the variate using any schemes - that is done by classes that extend it.

7. ***FixedPointFinderBrent*** customizes FixedPointFinderBracketing by applying the Brent's scheme of compound variate selector.
    - Brent's scheme, as implemented here, is described above.
    - This implementation retains absolute shifts that have happened to the variate for the past 2 iterations as the discriminant that determines the next variate to be generated.
    - ***FixedPointFinderBrent*** uses the following parameters specified in ***VariateIterationSelectorParams***:
        - The Variate Primitive that is regarded as the "fast" method
        - The Variate Primitive that is regarded as the "robust" method
        - The relative variate shift that determines when the "robust" method is to be invoked over the "fast"\
        - The lower bound on the variate shift between iterations that serves as the fall-back to the "robust"
8. ***FixedPointFinderZheng*** implements the root locator using Zheng's improvement to Brent's method.
    - It overrides the *iterateCompoundVariate* method in ***FixedPointFinderBrent*** to achieve the desired simplification in the iterative variate selection.

Software Framework Component - Iteration Entities

1. ***IteratedBracket*** holds the left/right bracket variates and the corresponding values for the ***ObjectiveFunction*** during each iteration.
2. ***IteratedVariate*** holds the variate and the corresponding value for the ***ObjectiveFunction*** during each iteration.
3. ***VariateIteratorPrimitive*** implements the various variate iterator primitives. It implements the following primitives:
        - Bisection

- False Position
- Quadratic
- Inverse Quadratic
- Ridder

It may be readily enhanced to accommodate additional primitives.

4. ***VariateIteratorSelectorParameters*** implements the control parameters for the compound variate selector scheme used in Brent's method.
   - Brent's method uses the following fields in ***VariateIteratorSelectorParameters*** to generate the next variate:
     - The Variate Primitive that is regarded as the "fast" method
     - The Variate Primitive that is regarded as the "robust" method
     - The relative variate shift that determines when the "robust" method is to be invoked over the "fast"
     - The lower bound on the variate shift between iterations that serves as the fallback to the "robust".

Software Framework Component – Initialization Heuristics

InitializationHeuristics implements several heuristics used to kick off the fixed point bracketing/search process. The following custom heuristics are implemented as part of the heuristics based kick-off:

- *Custom Bracketing Control Parameters*: Any of the standard bracketing control parameters can be customized to kick-off the bracketing search.
- *Soft Left/Right Bracketing Hints*: The left/right starting bracket edges are used as soft bracketing initialization hints.
- *Soft Mid Bracketing Hint*: A mid bracketing level is specified to indicate the soft bracketing kick-off.
- *Hard Bracketing Floor/Ceiling*: A pair of hard floor and ceiling limits are specified as a constraint to the bracketing.

- *Hard Search Boundaries*: A pair of hard left and right boundaries are specified to kick-off the final fixed point search.

These heuristics are further interpreted and developed inside the ***ExecutionInitializer*** and the ***ExecutionControl*** implementations.

Figure #1
Fixed Point Search SKU Flow

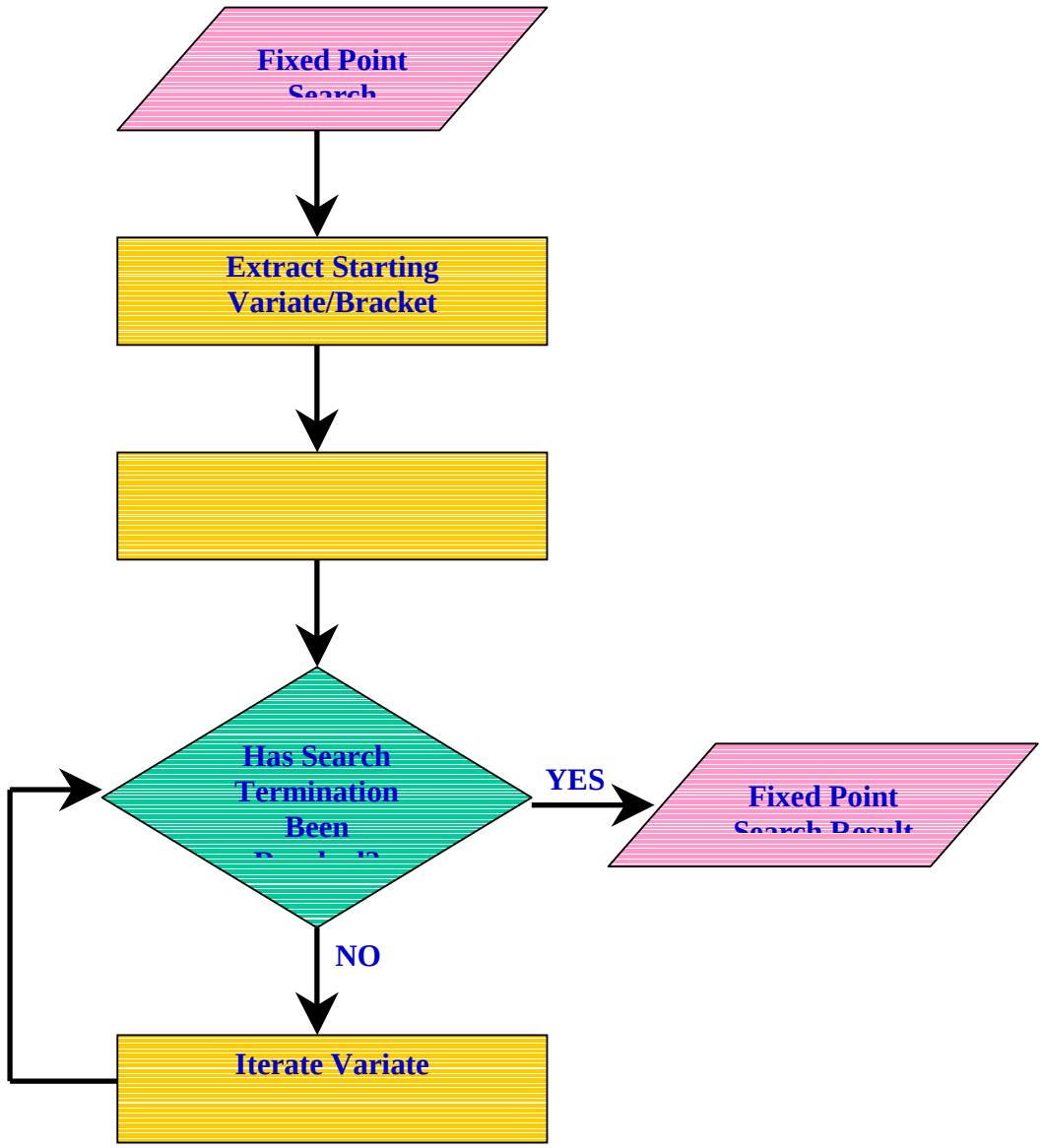

Figure #2
Bracketing SKU Flow

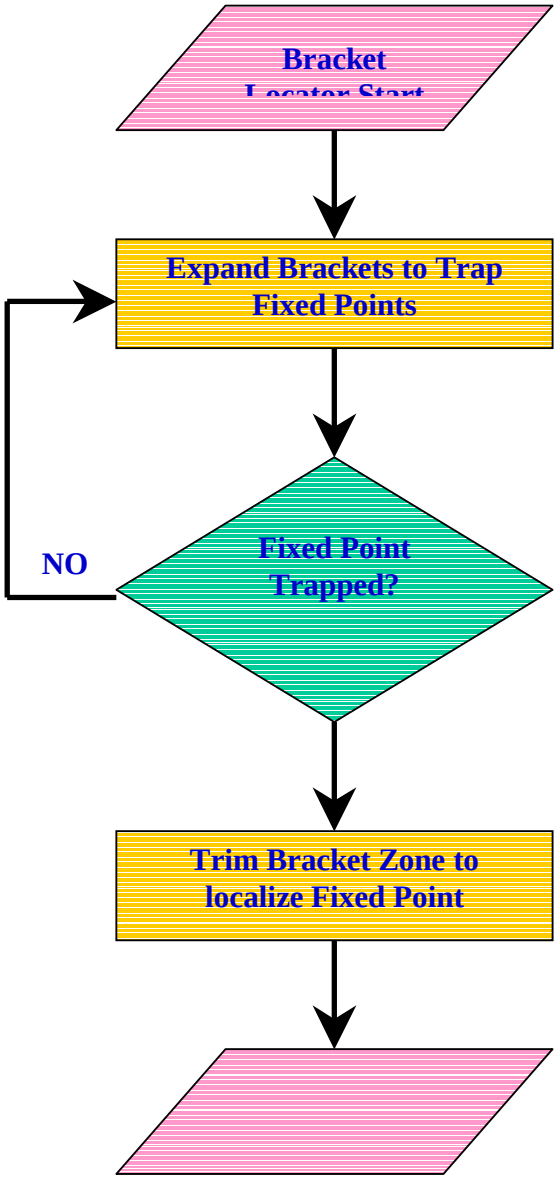

Figure #3
Objective Function Undefined at the Starting Variate

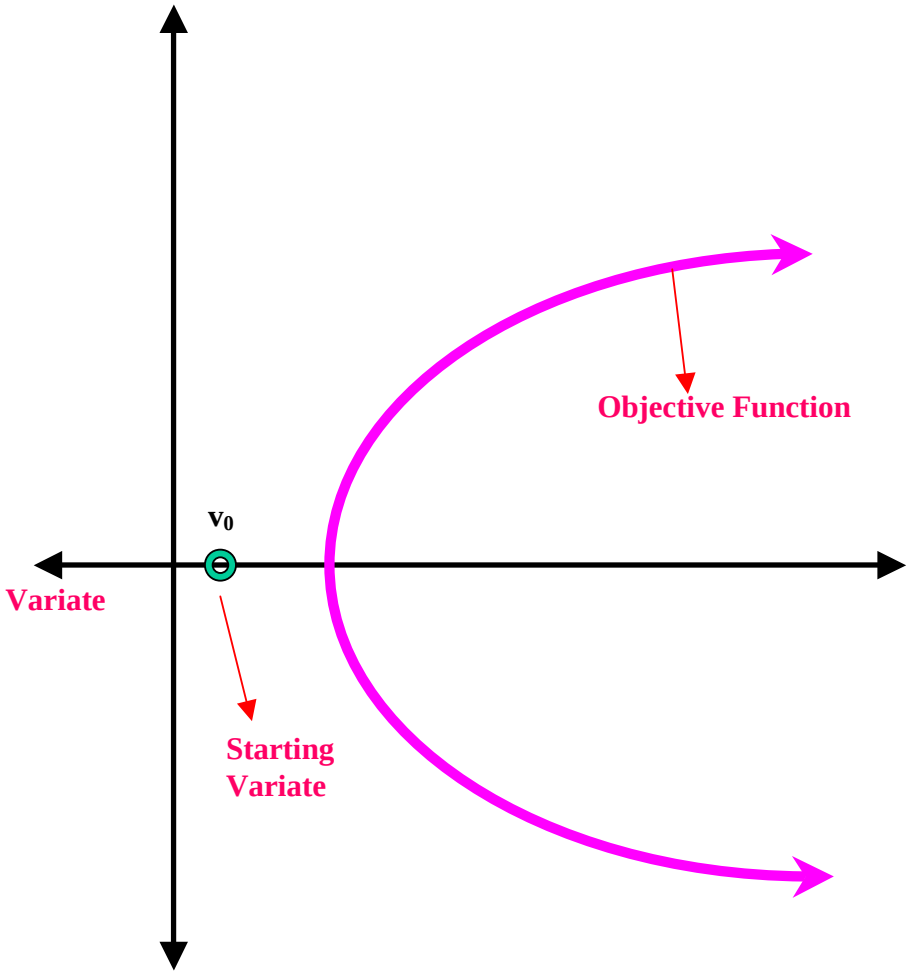

Figure #4
Objective Function Undefined at any of the Candidate Variates

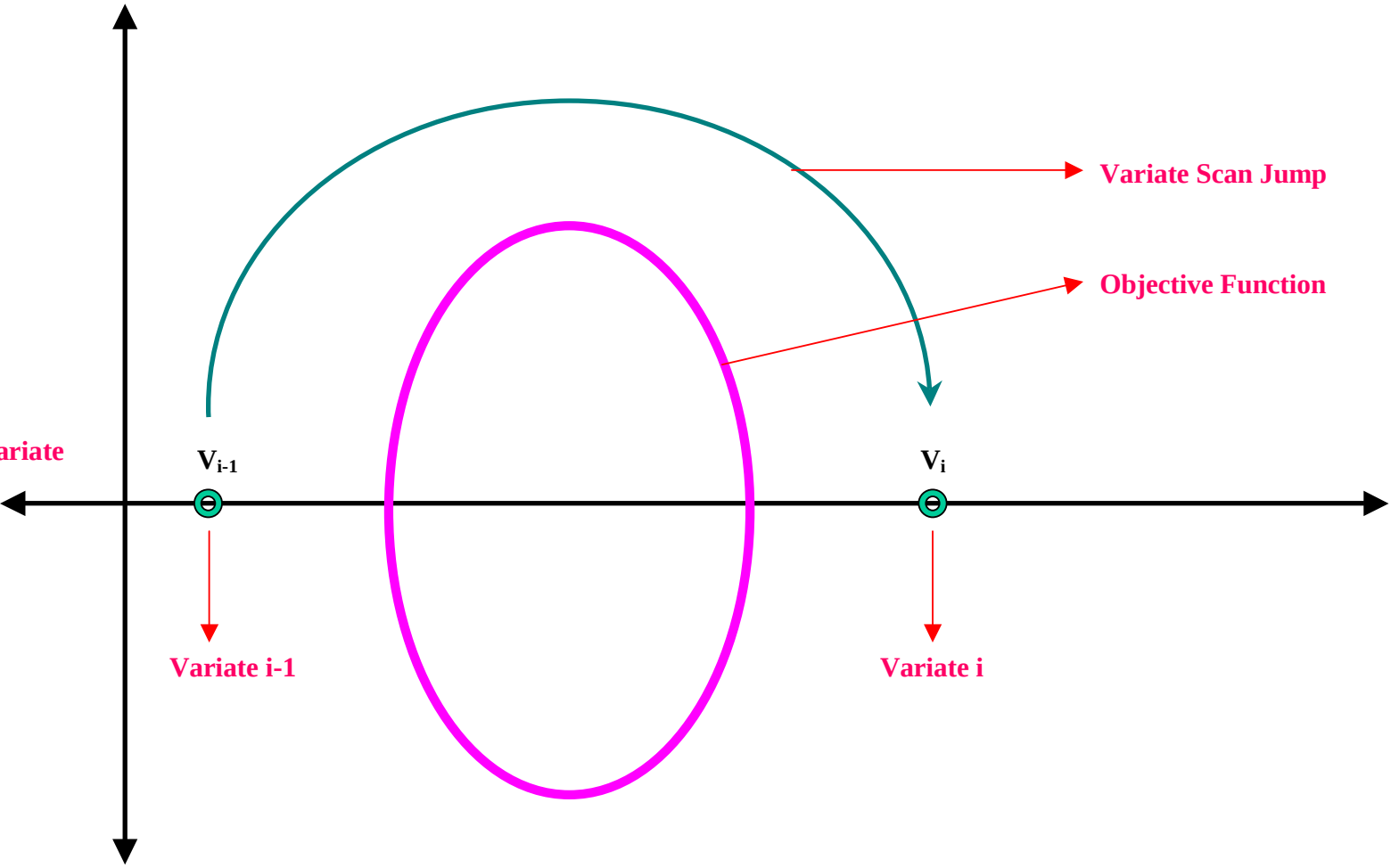

Figure #5
General Purpose Bracket Start Locator

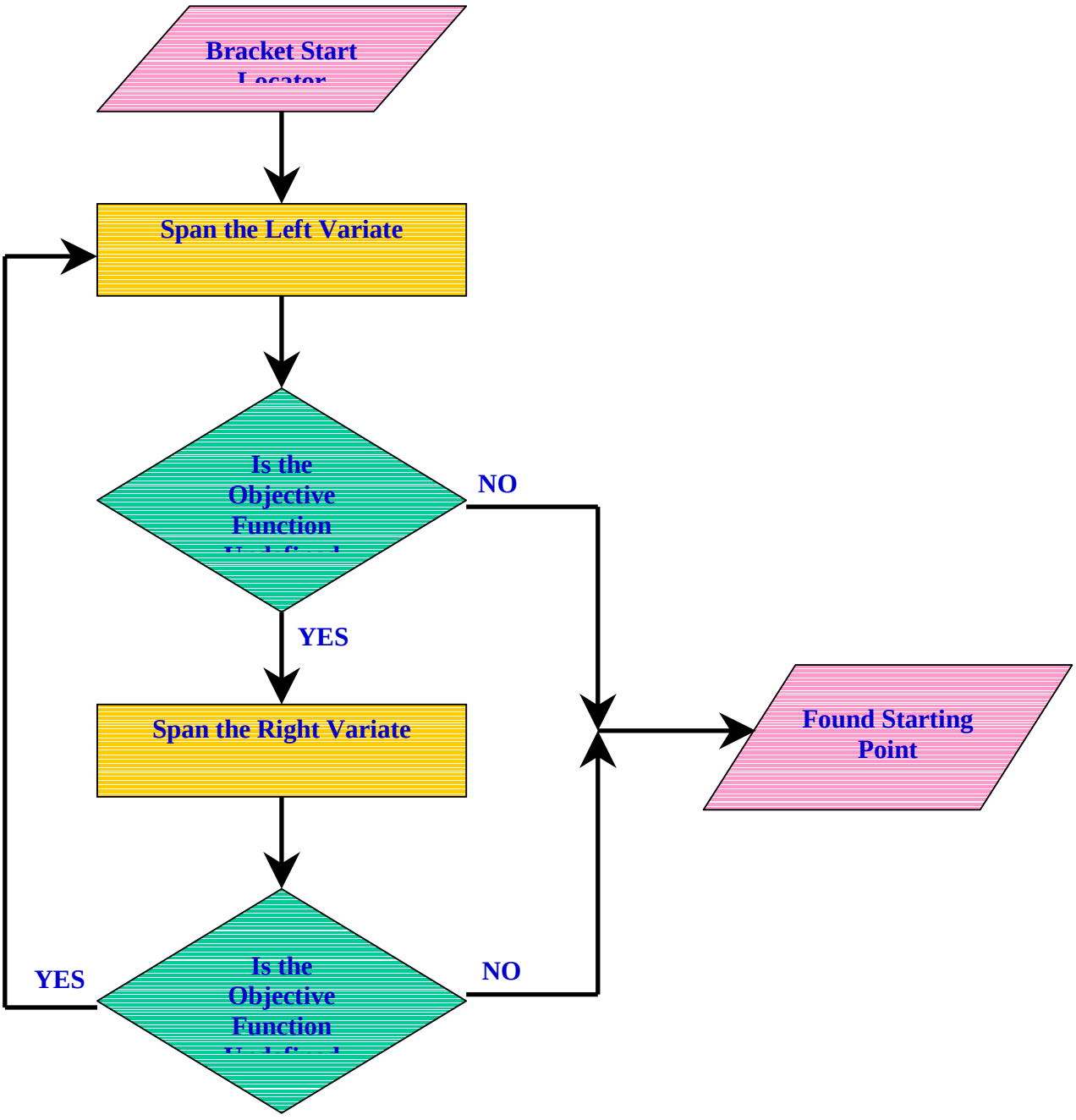

Figure #6
Bracketing when Objective Function Validity is Range-bound

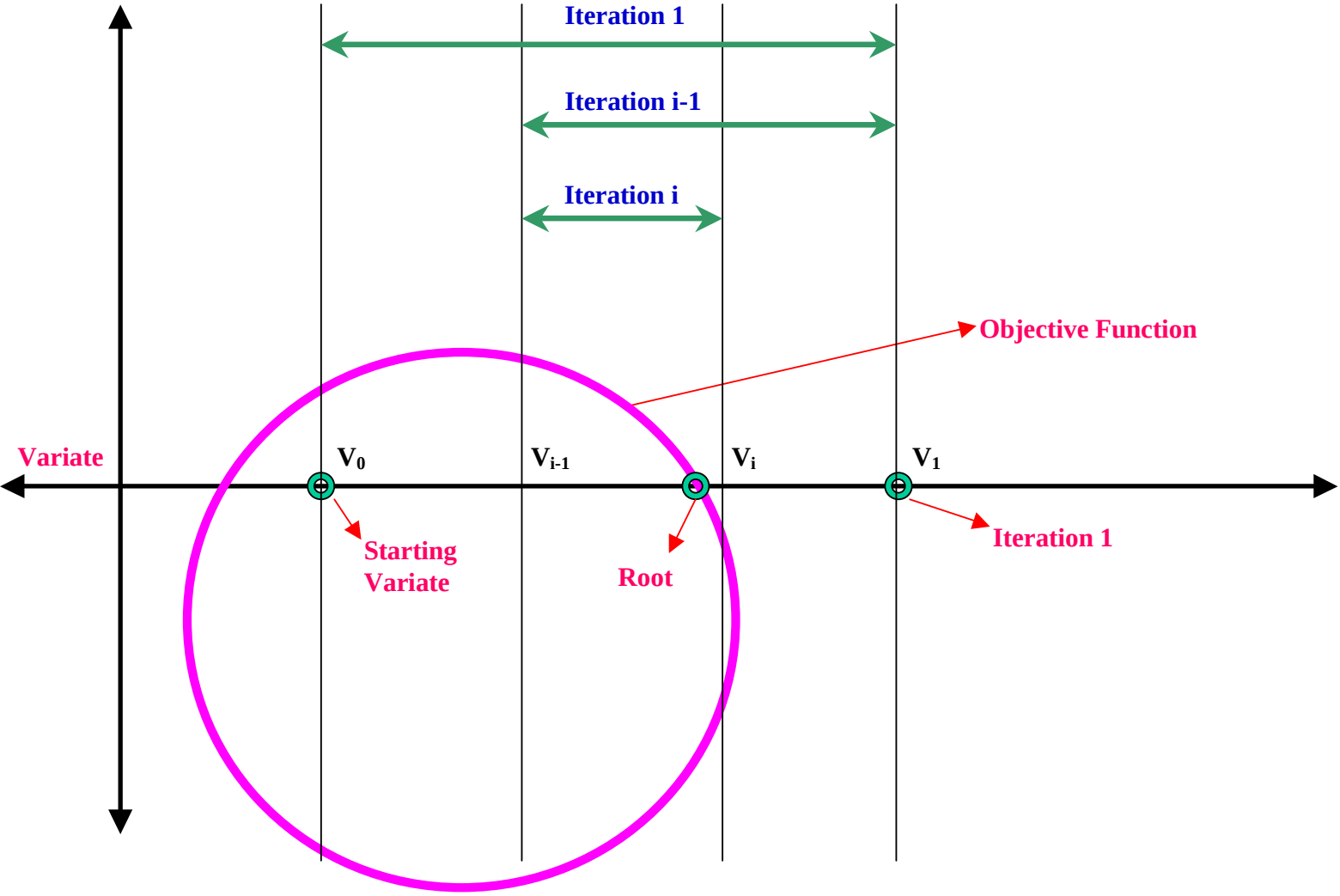

Figure #7
Objective Function Fixed Point Bracketing