Finding Roots of Algebraic and Transcendental Equations
Equations like this are called transcendental equations

\[ b \cdot x \cdot \cos x - \sin x = 0 \]

\[ bx = \tan x \]

Euler-buckling load for a fixed-pinned beam

\[ \tan x - x = 0 \]

has an infinite number of roots \( (x = 0, \pm 4.493, \pm 7.725, \ldots) \).

\[ Y = a \cosh (x/c) \], equation for a catenary

Solutions to these equations are always obtained iteratively. Starting point is really important for obtaining the proper solution. Lot of insight can be obtained from geometry and pictures.
Example: Natural Frequencies of cantilever

\[ f(\beta) = \cosh \beta \cos \beta + 1 = 0 \]

\[ \beta_i^4 = (2\pi f_i)^2 \frac{mL^3}{EI} \]

\( f_i = \) \( i \)th natural frequency (cps)

\( m = \) mass of the beam

\( L = \) length of the beam

\( E = \) modulus of elasticity

\( I = \) moment of inertia of the cross section
Find the solutions of \( f(x) = 0 \), where the function \( f \) is given

There are different ways of approaching a non-linear problem

The number of iterations required to reach the root depends largely on the intrinsic order of convergence of the method. Letting \( E_k \) be the error in the computed root after the \( k \)th iteration, an approximation of the error after the next iteration has the form

\[
E_{k+1} = c E_k^m \quad |c| < 1
\]

\( m \) is called the order of convergence. Note that for most of these methods the upper and lower bound for the root \([a, b]\) has to be given it is called as bracketing for obvious reasons.
Fixed Point Iteration method

If we want to find the solution to the equation $f(x) = 0$ re-write it in the form:

$$x = g(x).$$

we then iterate according the following rule:

$$x(i+1) = g(x(i)),$$

where $x(i+1)$ is the value Obtained after $i$th iteration. For functions with certain properties we Will always get a convergence to a **unique point** no matter what initial point we start with

We will demonstrate with a simple example.

Use fixed-point iteration to locate a root of:

$$f(x) = \exp(-x) - x$$

The function is separated in expressed in the form:

$$x_{i+1} = g(x_i) = \exp(-x_i)$$
\[ x_1 = g(x_0) = e^{-x_0} = e^0 = 1 \]

\[ x_2 = g(x_1) = e^{-x_1} = e^{-1} = 0.367898 \]

\[ x_3 = g(x_2) = e^{-x_2} = e^{-0.367898} = \ldots \]

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Approximate root</th>
<th>Approximate error</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>( x_i )</td>
<td>( e_A = \left</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.000000</td>
<td>100.0</td>
</tr>
<tr>
<td>2</td>
<td>0.367898</td>
<td>171.8</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>9</td>
<td>0.571143</td>
<td>1.93</td>
</tr>
<tr>
<td>10</td>
<td>0.564879</td>
<td>1.11</td>
</tr>
</tbody>
</table>

Error is obtained as relative error.
Graphical Representation

Solution of \(x^2 + 0.2 = x\)
Starting point: 0.65
Most basic method to bracket root: Iterative search method

It uses the property that when a function hits its root, the sign changes from +ve to -ve or vice-versa. So there has to be at least one root within that interval.
Problems with Iterative search besides being slow

Dx > 0.1: root will not be captured
Double-roots at $x = 1.0$ will not be obtainable.
The equation is $(x-1)^2 = 0$.
Certain singularities (poles) of \( f(x) \) can be mistaken for roots. For example, \( f(x) = \tan x \) changes sign at \( x = \pm \frac{1}{2} n \pi, \ n = 1, 3, 5, \ldots \), as shown in Fig. 4.1. However, these locations are not true zeros, since the function does not cross the
Matlab code for incremental search

```matlab
function [x1, x2] = rootsearch(f, a, b, dx)

% The function rootsearch looks for a zero of the function f(x) in the interval (a, b).
% The search starts at a and proceeds in steps dx toward b. Once a zero is detected,
% rootsearch returns its bounds (x1, x2) to the calling program. If a root was not
% detected, x1 = x2 = NaN is returned (in MATLAB NaN stands for “not a number”).
% After the first root (the root closest to a) has been bracketed, rootsearch can be
% called again with a replaced by x2 in order to find the next root. This can be repeated
% as long as rootsearch detects a root.
```

function [x1,x2] = rootsearch(func,a,b,dx)
% Incremental search for a root of f(x).
% USAGE: [x1,x2] = rootsearch(func,a,d,b,dx)
% INPUT:
% func = handle of function that returns f(x).
% a,b = limits of search.
% dx = search increment.
% OUTPUT:
% x1,x2 = bounds on the smallest root in (a,b);
% set to NaN if no root was detected

x1 = a;    f1 = feval(func,x1);
x2 = a + dx; f2 = feval(func,x2);
while f1*f2 > 0.0
    if x1 >= b
        x1 = NaN; x2 = NaN; return
    end
    x1 = x2;    f1 = f2;
x2 = x1 + dx; f2 = feval(func,x2);
end
Example

Use incremental search with $\Delta x = 0.2$ to bracket the smallest positive zero of $f(x) = x^3 - 10x^2 + 5$.

**Solution** We evaluate $f(x)$ at intervals $\Delta x = 0.2$, staring at $x = 0$, until the function changes its sign (value of the function is of no interest to us; only its sign is relevant). This procedure yields the following results:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$f(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>5.000</td>
</tr>
<tr>
<td>0.2</td>
<td>4.608</td>
</tr>
<tr>
<td>0.4</td>
<td>3.464</td>
</tr>
<tr>
<td>0.6</td>
<td>1.616</td>
</tr>
<tr>
<td>0.8</td>
<td>-0.888</td>
</tr>
</tbody>
</table>

From the sign change of the function, we conclude that the smallest positive zero lies between $x = 0.6$ and $x = 0.8$. 
Bisection Method

After a root of \( f(x) = 0 \) has been bracketed in the interval \((x_1, x_2)\), several methods can be used to close in on it. The method of bisection accomplishes this by successively halving the interval until it becomes sufficiently small. This technique is also known as the *interval halving method*. Bisection is not the fastest method available for computing roots, but it is the most reliable. Once a root has been bracketed, bisection will always close in on it.

The method of bisection uses the same principle as incremental search: if there is a root in the interval \((x_1, x_2)\), then \( f(x_1) \cdot f(x_2) < 0 \). In order to halve the interval, we compute \( f(x_3) \), where \( x_3 = \frac{1}{2}(x_1 + x_2) \) is the mid-point of the interval. If \( f(x_2) \cdot f(x_3) < 0 \), then the root must be in \((x_2, x_3)\) and we record this by replacing the original bound \( x_1 \) by \( x_3 \). Otherwise, the root lies in \((x_1, x_3)\), in which case \( x_2 \) is replaced by \( x_3 \). In either case, the new interval \((x_1, x_2)\) is half the size of the original interval. The bisection is repeated until the interval has been reduced to a small value \( \varepsilon \), so that

\[
|x_2 - x_1| \leq \varepsilon
\]
It is easy to compute the number of bisections required to reach a prescribed $\varepsilon$. The original interval $\Delta x$ is reduced to $\Delta x/2$ after one bisection, $\Delta x/2^2$ after two bisections, and after $n$ bisections it is $\Delta x/2^n$. Setting $\Delta x/2^n = \varepsilon$ and solving for $n$, we get

$$n = \frac{\ln (|\Delta x| / \varepsilon)}{\ln 2}$$
**bisect**

This function uses the method of bisection to compute the root of \( f(x) = 0 \) that is known to lie in the interval \((x_1, x_2)\). The number of bisections \( n \) required to reduce the interval to \( \text{tol} \) is computed from Eq. (4.1). The input argument \( \text{filter} \) controls the filtering of suspected singularities. By setting \( \text{filter} = 1 \), we force the routine to check whether the magnitude of \( f(x) \) decreases with each interval halving. If it does not, the “root” may not be a root at all, but a singularity, in which case \( \text{root} = \text{NaN} \) is returned. Since this feature is not always desirable, the default value is \( \text{filter} = 0 \).
function root = bisect(func,x1,x2,filter,tol)
% Finds a bracketed zero of f(x) by bisection.
% USAGE: root = bisect(func,x1,x2,filter,tol)
% INPUT:
% func    = handle of function that returns f(x).
% x1,x2   = limits on interval containing the root.
% filter  = singularity filter: 0 = off (default), 1 = on.
% tol     = error tolerance (default is 1.0e4*eps).
% OUTPUT:
% root    = zero of f(x), or NaN if singularity suspected.

if nargin < 5; tol = 1.0e4*eps; end
if nargin < 4; filter = 0; end
f1 = feval(func,x1);
if f1 == 0.0; root = x1; return; end
f2 = feval(func,x2);
if f2 == 0.0; root = x2; return; end
if f1*f2 > 0;
    error('Root is not bracketed in (x1,x2)')
end
n = ceil(log(abs(x2 - x1)/tol)/log(2.0));
for i = 1:n
    x3 = 0.5*(x1 + x2);
    f3 = feval(func, x3);
    if(filter == 1) & (abs(f3) > abs(f1))
        & (abs(f3) > abs(f2))
        root = NaN; return
    end
if f3 == 0.0
    root = x3; return
end
if f2*f3 < 0.0
    x1 = x3; f1 = f3;
else
    x2 = x3; f2 = f3;
end
end
root = (x1 + x2)/2;
Example

Find all the zeros of \( f(x) = x - \tan x \) in the interval \((0, 20)\) by the method of bisection. Utilize the functions rootsearch and bisect.

**Solution** Note that \( \tan x \) is singular and changes sign at \( x = \pi/2, 3\pi/2, \ldots \). To prevent bisect from mistaking these points for roots, we set filter = 1. The closeness of roots to the singularities is another potential problem that can be alleviated by using small \( \Delta x \) in rootsearch. Choosing \( \Delta x = 0.01 \), we arrive at the following program:
% Example 4.3 (root finding with bisection)
func = @(x) (x - tan(x));
a = 0.0; b = 20.0; dx = 0.01;

nroots = 0;
while 1
    [x1, x2] = rootsearch(func, a, b, dx);
    if isnan(x1)
        break
    else
        a = x2;
        x = bisect(func, x1, x2, 1);
        if ~isnan(x)
            nroots = nroots + 1;
            root(nroots) = x;
        end
    end
end

Running the program resulted in the output

>> root =

      0  4.4934  7.7253  10.9041  14.0662  17.2208
Geometric approach to finding roots of equations

Solution to the following equation when \( b = 4.0 \)

\[ b \cdot x \cdot \cos x - \sin x = 0 \]

A good iterative scheme should find all roots in a given bracket irrespective of initial guesses. This is a very tough task because different equations have very unpredictable behavior. This is especially true for higher dimensional equations.
Geometric and pictorial argument for solving the equation

\[ b \cdot x \cdot \cos x - \sin x = 0 \]

re-written as

\[ bx = \tan x \]

whose solutions occur at the intersections of the curves

\[ y = bx \quad \text{and} \quad y = \tan x \]

\[ y_1 = (0.4) \cdot (4.0) = 1.6 \quad 1.6 = \tan x_2 \quad \text{or} \quad x_2 = 1.012 + 3.142 = 4.154 \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>1.6</td>
</tr>
<tr>
<td>4.154</td>
<td>1.662</td>
</tr>
<tr>
<td>4.171</td>
<td>1.668</td>
</tr>
<tr>
<td>4.173</td>
<td>1.669</td>
</tr>
</tbody>
</table>
Analytically, our rapid convergence occurs because the two derivatives are small. Thus for the straight line

\[ \frac{dy}{dx} = b = 0.40 \]

while for the tangent curve

\[ \frac{dx}{dy} = \frac{1}{\frac{dy}{dx}} = \frac{1}{\sec^2 x} = 0.27 \]

and the convergence of the process is quite good.
What if we had used

\[ y = \tan x \quad \text{and} \quad x = \frac{y}{b} \]

We would have happily walked away from the real solution because the flatness criteria is absolutely not obeyed.

What if we had used \[ y = bx \cdot \cos x \quad \text{and} \quad y = \sin x \]

Computationally a bit cumbersome