Engineering Analysis ENG 3420 Fall 2009

Dan C. Marinescu
Office: HEC 439 B
Office hours: Tu-Th 11:00-12:00
Lecture 10

- Last time:
  - Bracketing vs Open Methods
  - Convergence vs Divergence
  - Simple Fixed-Point Iteration
  - Newton-Raphson

- Today:
  - More on functions nargin, nargout, varargin, and varargout
  - The secant open method
  - Optimization
    - Golden ratio → makes one-dimensional optimization efficient.
    - Parabolic interpolation → locate the optimum of a single-variable function.
    - `fminbnd` function → determine the minimum of a one-dimensional function.
    - `fminsearch` function → determine the minimum of a multidimensional function.

- Next Time
  - More on optimization
nargin \rightarrow \text{returns the number of input arguments specified for a function or -1 if the function has a variable number of input arguments.}
nargout \rightarrow \text{returns the number of output arguments specified for a function.}

Example: Function \textit{myplot} accepts an optional number of input and output arguments:

\begin{verbatim}
function [x0, y0] = myplot(x, y, npts, angle, subdiv)
  % The first two input arguments are required; the other three have default values. .
  if nargin < 5,
    subdiv = 20;
  end
  if nargin < 4,
    angle = 10;
  end
  if nargin < 3,
    npts = 25;
  end ...
  if nargout == 0
    plot(x, y)
  else
    x0 = x;
    y0 = y;
  end
\end{verbatim}
- `varargin` and `varargout` → used only inside a function M-file to contain the optional arguments to the function. Each must be declared as the last argument to a function, collecting all the inputs or outputs from that point onwards. In the declaration, `varargin` and `varargout` must be lowercase.

**Examples**

```matlab
function myplot(x,varargin) plot(x,varargin{:})
```
- collects all the inputs starting with the second input into the variable `varargin`.
- `myplot` uses the comma-separated list syntax `varargin{:}` to pass the optional parameters to plot.
- The call `myplot(sin(0:.1:1),'color',[.5 .7 .3],'linestyle',':')` results in `varargin` being a 1-by-4 cell array containing the values 'color', [.5 .7 .3], 'linestyle', and ':'.

```matlab
function [s,varargout] = mysize(x) nout = max(nargout,1)-1; s = size(x);
    for k=1:nout,
        varargout(k) = {s(k)};
    end
```
- returns the size vector and, optionally, individual sizes. So
  - `[s,rows,cols] = mysize(rand(4,5));`
Newton-Raphson Method

- Express $x_{i+1}$ function of $x_i$ and the values of the function and its derivative at $x_i$.

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

- Graphically → draw the tangent line to the $f(x)$ curve at some guess $x$, then follow the tangent line to where it crosses the $x$-axis.
function [root, relative_error, number_iterations] = newton_raphson(myfunction, derivative, initial_guess, desired_relative_error, max_number_iterations, varargin)

if (nargin < 3) error('at least 3 input arguments, required'); end
if (nargin < 4) is_empty(desired_relative_error), desired_relative_error = 0.0001; end
% set desired_relative_error to the default, 0.0001, if none specified
if (nargin < 5) is_empty(max_number_iterations), max_number_iterations = 50; end
% set max_number_iterations to the default, 50, if none specified

number_iterations = 0;
current_guess = initial_guess;

while (1)
    next_guess = current_guess;
current_guess = current_guess - (myfunction(current_guess)/derivative(current_guess));
number_iterations = number_iterations + 1;
% fprintf(' %6.0f number_iterations', number_iterations);

    if (current_guess ~= 0.0)
        relative_error = abs(((current_guess - next_guess)/current_guess)*100);
    end
if (relative_error <= desired_relative_error) | (number_iterations >= max_number_iterations),
    break,
end
fprintf('iteration= %2.0f  guess= %8.5f  relative_error=%8.5f  
', number_iterations, current_guess, relative_error);
end
root = current_guess;
Example – fast converging case \( f(x) = \exp(-x) - x \)

\[
\begin{align*}
&\text{>> fun=@(x) exp(-x)-x; der=@(x) -exp(-x)-1; } \\
&[\text{root,rel_error,niter}]=\text{newton_raphson}(\text{fun,der,0.0,0.000001,500}); \text{ root, rel_error,niter} \\
&\text{iteration= 1  guess= 0.50000  relative_error=100.00000} \\
&\text{iteration= 2  guess= 0.56631  relative_error=11.70929} \\
&\text{iteration= 3  guess= 0.56714  relative_error= 0.14673} \\
&\text{iteration= 4  guess= 0.56714  relative_error= 0.00002} \\
&\text{root = 0.5671} \\
&\text{rel_error =5.0897e-013} \\
&\text{niter = 5}
\end{align*}
\]
Slow converging case: $f(x) = x^{10} - 1$ init_guess=0.5

```matlab
>> fun=@(x) x^(10)-1; der=@(x) 10*x^(9); init_guess=0.5;
[root,rel_error,niter]=newton_raphson(fun,der,init_guess,0.000001,500); root,
rel_error,niter

iteration=  1  guess= 51.65000 relative_error=99.03195
iteration=  2  guess= 46.48500 relative_error=11.11111
iteration=  3  guess= 41.83650 relative_error=11.11111
iteration=  4  guess= 37.65285 relative_error=11.11111
iteration=  5  guess= 33.88757 relative_error=11.11111
iteration=  6  guess= 30.49881 relative_error=11.11111
iteration=  7  guess= 27.44893 relative_error=11.11111
iteration=  8  guess= 24.70403 relative_error=11.11111
iteration=  9  guess= 22.23363 relative_error=11.11111
iteration= 10  guess= 20.01027 relative_error=11.11111
iteration= 11  guess= 18.00924 relative_error=11.11111
iteration= 12  guess= 16.20832 relative_error=11.11111
iteration= 13  guess= 14.58749 relative_error=11.11111
iteration= 14  guess= 13.12874 relative_error=11.11111
iteration= 15  guess= 11.81586 relative_error=11.11111
iteration= 16  guess= 10.63428 relative_error=11.11111
iteration= 17  guess=  9.57085 relative_error=11.11111
iteration= 18  guess=  8.61376 relative_error=11.11111
iteration= 19  guess=  7.75239 relative_error=11.11111
```
iteration= 20  guess=  6.97715 relative_error=11.11111
iteration= 21  guess=  6.27943 relative_error=11.11111
iteration= 22  guess=  5.65149 relative_error=11.11111
iteration= 23  guess=  5.08634 relative_error=11.11111
iteration= 24  guess=  4.57771 relative_error=11.11111
iteration= 25  guess=  4.11994 relative_error=11.11111
iteration= 26  guess=  3.70794 relative_error=11.11110
iteration= 27  guess=  3.33715 relative_error=11.11109
iteration= 28  guess=  3.00344 relative_error=11.11104
iteration= 29  guess=  2.70310 relative_error=11.11090
iteration= 30  guess=  2.43280 relative_error=11.11052
iteration= 31  guess=  2.18955 relative_error=11.10941
iteration= 32  guess=  1.97069 relative_error=11.10624
iteration= 33  guess=  1.77384 relative_error=11.09714
iteration= 34  guess=  1.59703 relative_error=11.07110
iteration= 35  guess=  1.43881 relative_error=10.99684
iteration= 36  guess=  1.29871 relative_error=10.78735
iteration= 37  guess=  1.17835 relative_error=10.21396
iteration= 38  guess=  1.08335 relative_error= 8.76956
iteration= 39  guess=  1.02366 relative_error= 5.83053
iteration= 40  guess=  1.00232 relative_error= 2.12993
iteration= 41  guess=  1.00002 relative_error= 0.22920
iteration= 42  guess=  1.00000 relative_error= 0.00239

root =  1
rel_error = 2.5776e-007
niter = 43
The same function but a better initial guess

```
fun=@(x) x^(10)-1; der=@(x) 10*x^(9); init_guess=1.1;
[root,rel_error,niter]=newton_raphson(fun,der,init_guess,0.000001,500); root, rel_error,niter

iteration= 1 guess= 1.03241 relative_error= 6.54684
iteration= 2 guess= 1.00422 relative_error= 2.80760
iteration= 3 guess= 1.00008 relative_error= 0.41363
iteration= 4 guess= 1.00000 relative_error= 0.00787
iteration= 5 guess= 1.00000 relative_error= 0.00000

root = 1
rel_error = 3.5527e-013
niter = 6
```
Pros and Cons

- Pro: The error of the \(i+1\)th iteration is roughly proportional to the square of the error of the \(i\)th iteration - this is called *quadratic convergence*.
- Con: Some functions show slow or poor convergence.
Example

- Find the weight $m$ of a bungee jumper which reaches a velocity of 36 m/s after 4 seconds when the drag coefficient is $c = 0.25$ kg/m.
- The function to evaluate and its derivative are:

\[
f(m) = \sqrt{\frac{g m}{c}} \tanh \left( \sqrt{\frac{g c}{m}} t \right) - v(t)
\]

\[
\frac{df(m)}{dm} = \frac{1}{2} \sqrt{\frac{g}{mc}} \tanh \left( \sqrt{\frac{g c}{m}} t \right) - \frac{g}{2m} t \sec h^2 \left( \sqrt{\frac{g c}{m}} t \right)
\]
Example (cont’d)

init_guess=140;
fun=@(m) sqrt(9.81*m/0.25)*tanh(sqrt((9.81*0.25/m)*4)-36);
der=@(m) 0.5* sqrt(9.81/(m*0.25))*tanh(sqrt(9.81*0.25/m)*4)-
     (9.81/2*m)*4*(sech(sqrt(9.81*0.25/m)*4))^2;
[root,rel_error,niter]=newton_raphson(fun,der,init_guess,0.01,5000); root,
rel_error,niter
Secant Methods

- The evaluation of the derivative could pose problems for the Newton-Raphson method.
- The derivatives of some functions may be difficult or inconvenient to evaluate. For these cases, the derivative can be approximated by a backward finite divided difference:

\[ f'(x_i) \approx \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i} \]

- Substitution of this approximation for the derivative to the Newton-Raphson method equation gives:

\[ x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)} \]

- This method requires two initial estimates of \( x \) but does not require an analytical expression of the derivative.
fzero built-in function

fzero \to\text{ applies both bracketing methods and open methods.}

- Using an initial guess:
  \[ x = \text{fzero}(function, x0) \]
  \[ [x, fx] = \text{fzero}(function, x0) \]
  - \textit{function} \to\text{ function handle to the function being evaluated}
  - \textit{x0} \to\text{ the initial guess}
  - \textit{x} \to\text{ the location of the root}
  - \textit{fx} \to\text{ is the function evaluated at that root}

- Using an initial bracket:
  \[ x = \text{fzero}(function, [x0 \ x1]) \]
  \[ [x, fx] = \text{fzero}(function, [x0 \ x1]) \]
  - \textit{x0} and \textit{x1} are guesses that \textit{must} bracket a sign change
Example

- \[ [x, \, fx] = \text{fzero}(@(x) \, x^{10}-1, \, 0.5) \] \% Use fzero to find roots of \( f(x)=x^{10}-1 \) starting with an initial guess of \( x=0.5 \).

\[
[x, \, fx] = \text{fzero}(@(x) \, x^{10}-1, \, 0.5) \\
x = 1 \\
fx = 0 \\
\text{after 35 calls}
\]
fzero options

Options may be passed to `fzero` as a third input argument - the options are a data structure created by the `optimset` command

```
options = optimset('par1', val1, 'par2', val2, ...)
```

- `par_n` → the name of the parameter to be set
- `val_n` is the value to which to set that parameter
- The parameters commonly used with fzero are:
  - `display`: when set to ‘iter’ displays a detailed record of all the iterations
  - `tolx`: A positive scalar that sets a termination tolerance on x.
options = optimset('display', 'iter');
[x, fx] = fzero(@(x) x^10-1, 0.5, options)

Search for an interval around 0.5 containing a sign change:

<table>
<thead>
<tr>
<th>Func-count</th>
<th>a</th>
<th>f(a)</th>
<th>b</th>
<th>f(b)</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5</td>
<td>-0.999023</td>
<td>0.5</td>
<td>-0.999023</td>
<td>initial interval</td>
</tr>
<tr>
<td>3</td>
<td>0.485858</td>
<td>-0.999267</td>
<td>0.514142</td>
<td>-0.998709</td>
<td>search</td>
</tr>
<tr>
<td>5</td>
<td>0.48</td>
<td>-0.999351</td>
<td>0.52</td>
<td>-0.998554</td>
<td>search</td>
</tr>
<tr>
<td>7</td>
<td>0.471716</td>
<td>-0.999454</td>
<td>0.528284</td>
<td>-0.998307</td>
<td>search</td>
</tr>
<tr>
<td>9</td>
<td>0.46</td>
<td>-0.999576</td>
<td>0.54</td>
<td>-0.997892</td>
<td>search</td>
</tr>
<tr>
<td>11</td>
<td>0.443431</td>
<td>-0.999706</td>
<td>0.556569</td>
<td>-0.997148</td>
<td>search</td>
</tr>
<tr>
<td>13</td>
<td>0.42</td>
<td>-0.999829</td>
<td>0.58</td>
<td>-0.995692</td>
<td>search</td>
</tr>
<tr>
<td>15</td>
<td>0.386863</td>
<td>-0.999925</td>
<td>0.613137</td>
<td>-0.992491</td>
<td>search</td>
</tr>
<tr>
<td>17</td>
<td>0.34</td>
<td>-0.999979</td>
<td>0.66</td>
<td>-0.984317</td>
<td>search</td>
</tr>
<tr>
<td>19</td>
<td>0.273726</td>
<td>-0.999998</td>
<td>0.726274</td>
<td>-0.959167</td>
<td>search</td>
</tr>
<tr>
<td>21</td>
<td>0.18</td>
<td>-1</td>
<td>0.82</td>
<td>-0.862552</td>
<td>search</td>
</tr>
<tr>
<td>23</td>
<td>0.0474517</td>
<td>-1</td>
<td>0.952548</td>
<td>-0.385007</td>
<td>search</td>
</tr>
<tr>
<td>25</td>
<td>-0.14</td>
<td>-1</td>
<td>1.14</td>
<td>2.70722</td>
<td>search</td>
</tr>
</tbody>
</table>

Search for a zero in the interval [-0.14, 1.14]:

<table>
<thead>
<tr>
<th>Func-count</th>
<th>x</th>
<th>f(x)</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>-0.14</td>
<td>-1</td>
<td>initial</td>
</tr>
<tr>
<td>26</td>
<td>0.205272</td>
<td>-1</td>
<td>interpolation</td>
</tr>
<tr>
<td>27</td>
<td>0.672636</td>
<td>-0.981042</td>
<td>bisection</td>
</tr>
<tr>
<td>28</td>
<td>0.906318</td>
<td>-0.626056</td>
<td>bisection</td>
</tr>
<tr>
<td>29</td>
<td>1.02316</td>
<td>0.257278</td>
<td>bisection</td>
</tr>
<tr>
<td>30</td>
<td>0.989128</td>
<td>-0.103551</td>
<td>interpolation</td>
</tr>
<tr>
<td>31</td>
<td>0.998894</td>
<td>-0.0110017</td>
<td>interpolation</td>
</tr>
<tr>
<td>32</td>
<td>1.00001</td>
<td>7.68385e-005</td>
<td>interpolation</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>-3.83061e-007</td>
<td>interpolation</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>-1.3245e-011</td>
<td>interpolation</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>0</td>
<td>interpolation</td>
</tr>
</tbody>
</table>

Zero found in the interval [-0.14, 1.14]

x = 1 fx = 0
Polynomials

- Roots → built in function to determine all the roots of a polynomial - including imaginary and complex ones.

- `x = roots(c)`
  - `x` is a column vector containing the roots
  - `c` is a row vector containing the polynomial coefficients

Example:

- Find the roots of
  \[ f(x) = x^5 - 3.5x^4 + 2.75x^3 + 2.125x^2 - 3.875x + 1.25 \]

  ```matlab
  >> x = roots([1 -3.5 2.75 2.125 -3.875 1.25])
  >> x =
  2.0000
  -1.0000
  1.0000 + 0.5000i
  1.0000 - 0.5000i
  0.5000
  ```
poly and polyval builtin functions

- poly \(\to\) determine polynomial coefficients if roots are given:
  - \(b = \text{poly}([0.5 -1])\)
  - Finds \(f(x)\) where \(f(x) = 0\) for \(x=0.5\) and \(x=-1\)
  - MATLAB reports \(b = [1.0000 0.5000 -0.5000]\)
  - This corresponds to \(f(x)=x^2+0.5x-0.5\)

- polyval \(\to\) evaluate a polynomial at one or more points:
  - \(a = [1 -3.5 2.75 2.125 -3.875 1.25]\);
  - If used as coefficients of a polynomial, this corresponds to \(f(x)=x^5-3.5x^4+2.75x^3+2.125x^2-3.875x+1.25\)
  - \(\text{polyval}(a, 1)\)
  - This calculates \(f(1)\), which MATLAB reports as -0.2500
Optimization

- Critical for solving engineering and scientific problems.
  - One-dimensional versus multi-dimensional optimization.
  - Global versus local optima.
  - A maximization problem can be solved with a minimizing algorithm.

- Optimization is a hard problem when the search space for the optimal solution is very large. Heuristics such as simulated annealing, genetic algorithms, neural networks.

- Algorithms
  - Golden ratio $\phi$ makes one-dimensional optimization efficient.
  - Parabolic interpolation $\triangledown$ locate the optimum of a single-variable function.
  - `fminbnd` function $\rightarrow$ determine the minimum of a one-dimensional function.
  - `fminsearch` function $\rightarrow$ determine the minimum of a multidimensional function.

- How to develop contours and surface plots to visualize two-dimensional functions.
Optimization

- Find the most effective solution to a problem subject to a certain criteria.
- Find the maxima and/or minima of a function of one or more variables.
One- versus multi-dimensional optimization

- One-dimensional problems involve functions that depend on a single dependent variable - for example, $f(x)$.
- Multidimensional problems involve functions that depend on two or more dependent variables - for example, $f(x,y)$
Global versus local optimization

- **Global optimum** → the very best solution.
- **Local optimum** → solution better than its immediate neighbors. Cases that include local optima are called *multimodal*.
- Generally we wish to find the global optimum.
One- versus Multi-dimensional Optimization

- One-dimensional problems involve functions that depend on a single dependent variable - for example, $f(x)$.
- Multidimensional problems involve functions that depend on two or more dependent variables - for example, $f(x,y)$.
Golden-Section Search

- Algorithm for finding a minimum on an interval \([x_l, x_u]\) with a \textit{single} minimum (\textit{unimodal} interval); uses the \textit{golden ratio} \(\phi=1.6180\) to determine location of two interior points \(x_1\) and \(x_2\):

\[
d = (\phi - 1)(x_u - x_l)
\]

\[
x_1 = x_l + d
\]

\[
x_2 = x_u - d
\]

- One of the interior points can be re-used in the next iteration.
  - \(f(x_1) < f(x_2) \rightarrow x_2\) will be the new lower limit and \(x_1\) the new \(x_2\).
  - \(f(x_2) < f(x_1) \rightarrow x_1\) will be the new upper limit and \(x_2\) the new \(x_1\).
- $f(x_1) < f(x_2) \Rightarrow x_2$ is the new lower limit and $x_1$ the new $x_2$.

- $f(x_2) < f(x_1) \Rightarrow x_1$ is the new upper limit and $x_2$ the new $x_1$. 
function [x,fx,ea,iter]=goldmin(f,xl,xu,es,maxit,varargin)
% goldmin: minimization golden section search
% [xopt,fopt,ea,iter]=goldmin(f,x1,xu,es,maxit,p1,p2,...): 
% uses golden section search to find the minimum of f 
% input:
% f = function handle
% xl, xu = lower and upper guesses
% es = desired relative error (default = 0.0001)
% maxit = maximum allowable iterations (default = 50)
% p1,p2,... = additional parameters used by f
% output:
% x = location of minimum
% fx = minimum function value
% ea = approximate relative error (%)
% iter = number of iterations

if nargin<3,error('at least 3 input arguments required'),end
if nargin<4||isempty(es), es=0.0001;end
if nargin<5||isempty(maxit), maxit=50;end
phi=(1+sqrt(5))/2;
iter=0;
while(1)
    d = (phi-1)*(xu - xl);
    x1 = xl + d;
    x2 = xu - d;
    if f(x1,varargin()) < f(x2,varargin())
        xopt = x1;
        xl = x2;
    else
        xopt = x2;
        xu = x1;
    end
    iter=iter+1;
    if xopt==0, ea = (2 - phi) * abs((xu - xl) / xopt) * 100;end
    if ea <= es || iter >= maxit,break,end
end
x=xopt;fx=f(xopt,varargin());
Parabolic interpolation

- Parabolic interpolation requires three points to estimate optimum location.
- The location of the maximum/minimum of a parabola defined as the interpolation of three points \((x_1, x_2, \text{ and } x_3)\) is:

\[
x_4 = x_2 - \frac{1}{2} \frac{(x_2 - x_1)^2 [f(x_2) - f(x_3)] - (x_2 - x_3)^2 [f(x_2) - f(x_1)]}{(x_2 - x_1)[f(x_2) - f(x_3)] - (x_2 - x_3)[f(x_2) - f(x_1)]}
\]

- The new point \(x_4\) and the two surrounding it (either \(x_1\) and \(x_2\) or \(x_2\) and \(x_3\)) are used for the next iteration of the algorithm.
fminbnd built-in function

- **fminbnd** → combines the golden-section search and the parabolic interpolation.
- Example
  - \[[x_{\text{min}}, f_{\text{val}}] = \text{fminbnd}(\text{function}, x1, x2)\]
- Options may be passed through a fourth argument using \text{optimset}, similar to \text{fzero}.
**fminsearch** built-in function

- **fminsearch** → determine the minimum of a multidimensional function.
  - \([x_{min}, f_{val}] = fminsearch(function, x0)\)
  - \(x_{min}\) → a row vector containing the location of the minimum
  - \(x0\) → an initial guess; must contain as many entries as the function expects.
- The function must be written in terms of a single variable, where different dimensions are represented by different indices of that variable.
Example: minimize \( f(x,y)=2+x-y+2x^2+2xy+y^2 \)

- **Step 1:** rewrite as:
  \[
  f(x_1, x_2)=2+x_1-x_2+2(x_1)^2+2x_1x_2+(x_2)^2
  \]

- **Step 2:** define the function \( f \) using Matlab syntax:
  \[
  f=@(x) 2+x(1)-x(2)+2*x(1)^2+2*x(1)*x(2)+x(2)^2
  \]

- **Step 3:** invoke \texttt{fminsearch}
  \[
  [x, fval] = \texttt{fminsearch}(f, [-0.5, 0.5])
  \]
  \( x_0 \) has two entries - \( f \) is expecting it to contain two values.
  the minimum value is 0.7500 at a location of \([-1.000 1.5000]\)
Heuristics for global optimization

- Global optimization is a very hard problem when the search space for the solution is very large.

- Heurisitic → adjective for experience-based techniques that help in problem solving, learning and discovery. A heuristic method is particularly used to rapidly come to a solution that is hoped to be close to the best possible answer, or 'optimal solution'.

- Heuristics → noun meaning "rules of thumb", educated guesses, intuitive judgments or simply common sense.

- Heuristics for global optimization
  - Simulated annealing
  - Genetic algorithms
  - Neural networks
Simulated annealing (SA)

- **Inspired from metallurgy:**
  - Annealing is a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects.
  - The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one.

- **Each step of the SA algorithm:**
  - Replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter $T$ (called the *temperature*), that is gradually decreased during the process.
  - The dependency is such that the current solution changes almost randomly when $T$ is large, but increasingly "downhill" as $T$ goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local minima—which are the bane of greedier methods.
Genetic algorithms

- Global search heuristics to find exact or approximate solutions to optimization and search problems. Genetic algorithms are a particular class of evolutionary algorithms (EA) that use evolutionary biology concepts such as inheritance, mutation, selection, and crossover.

- The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.
Neural networks

- **Biological neural networks** → are made up of real biological neurons that are connected or functionally related in the peripheral nervous system or the central nervous system. In the field of neuroscience, they are often identified as groups of neurons that perform a specific physiological function in laboratory analysis.

- **Artificial neural networks** → are made up of interconnecting artificial neurons (programming constructs that mimic the properties of biological neurons). Artificial neural networks may either be used to gain an understanding of biological neural networks, or for solving artificial intelligence problems without necessarily creating a model of a real biological system.
Multidimensional Visualization

- Functions of two-dimensions may be visualized using contour or surface/mesh plots.
Multidimensional Visualization

- Functions of two-dimensions may be visualized using contour or surface/mesh plots.