# Euler, ‘our jewel’, Numerical Methods, Graphs & Optimisation

Gavin Conran

# Introduction

This revision / background paper introduces the Swiss mathematician, physicist, astonomer and engineer, ***Leonard Euler***, viewed by many as the finest mathematian who ever lived and, to make him familiar with computer scientists, the creator of the mathematical notaion to describe a function, f(x). The paper starts with a discussion on what Richard Feynman refered to as “our jewel”, aka ***Euler’s Formula***, which establishes the fundamental relationship between the Trigonometric Functions and the Complex Exponential Function. It may be supposition, but it could be imagined that Euler may indirectly have motivated Feynman in his discovery of the relationship between PDEs and probability theory expressed in his Feynman-Kac formula. Euler’s Formula is a mathematical foundation stone of Differential Equations and Digital Signal Processing, both of which are explored, in some detail, in the Fourier paper.

The discussion moves onto Euler’s central role in ***Numerical Methods*** to solve Ordinary, Partial and Stochastic Differential Equations with an exploration of ***Euler’s Method***, and compares the difference between function approximation using Taylor and Fourier Series expansions. This is followed by a brief discussion on Runge-Kutta, a higher order numerical method, which is a more accurate approximation technique than a Taylor Expansion.The section ends with a short discussion on error analysis for time-stepping routines.

We switch to Euler’s ‘seven bridges of Königsburg’ and his key role in ***Graph Theory*** and then hand over to more modern day Graph algorithms. We start by taking a look at two key problems: the first is finding the Minimum Spanning Tree of a Graph and the second is computing the Shortest Path between two nodes (vertices) of a Graph. We then go on to discuss how identifying negative cycles in Graphs, via the Bellman-Ford algorithm, is used to identify Arbitrage in finacial markets.

The final section concentrates on ***Optimisation***. Starting with ***continuous*** optimisation we investigate a number of Machine Learning algorithms; mainly Stochastic Gradient Descent (SGD) with Back-Propagation used in the ArtNet project, and Expectation-Maximisation (EM) used in unsupervised learning, especially clustering. We finish continuous optimisation with an exploration of Lagrange Multipliers, a mathematical technique used in optimisation, especially in Finance and Economics, with an example of the Modern Portfolio Theory.

We then move on to ***discrete*** optimisation where we explore solutions to the Max Flow problem through the prisms of Graphs and Linear Programming high-lighting the realtionship between Graph algorithms and Linear Programming through the notions of ***Reduction*** and ***Intractability***.

The section, and indeed the paper, ends with a return to ***Stochastic processes*** and Monte Carlo methods, in the guise of Simulated Annealing, an optimisation technique based on the Hastings-Metropolis algorithm. As a coded example, the ***NP-complete*** problem of the Travelling Salesman Problem is solved using simulated annealing.

# Euler's Formula

*(Feynman refered to it as "our jewel")*

Establishes the fundamental relationship between the Trigonometric Functions and the Complex Exponential Function.

For any real number x, ***Euler's Formula*** states:

$$e^{ix} = \cos(x) + i \cdot \sin(x)$$

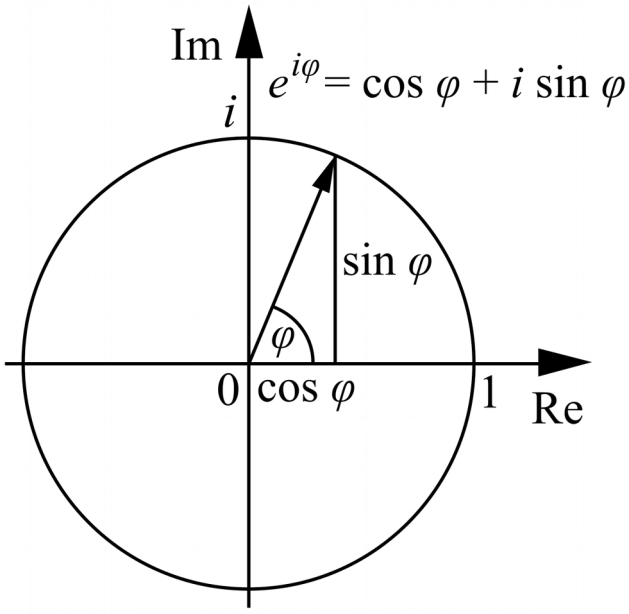
where **e**, Euler's Number, is the base of the natural logarithm
**i**: imaginary unit
**cos / sin** are the trig functions
**x** is given in Radians

***Euler's Identity*** is when $x = \pi$ and Euler's Formula evaluates to:

$$e^{ix} + 1 = 0$$

## Applications in Complex Number Theory

- $e^{i\varphi} + 1 = 0$ is a Unit Complex Number, i.e. it traces out the Unit Circle in the Complex Plane as $\varphi$ ranges through the real numbers, in Radians.
- A Point on the Complex Plane can be represented by a Complex Number written in Cartesian Coordinates
- Euler's Formula provides a means of conversion between Cartesian and Polar Coordinates




$$z = x + iy = |z| \cdot (\cos\varphi + i\sin\varphi) = r\,e^{i\varphi}$$
$$\bar{z} = x - iy = |z| \cdot (\cos\varphi - i\sin\varphi) = r\,e^{-i\varphi}$$

Where

$x = \Re\, z$

$y = \Im\, z$

$r = |z| = \sqrt{(x^2 + y^2)}$

$\varphi = arg\, z = atan2(y, x)$ , the angle between the x axis and the vector z

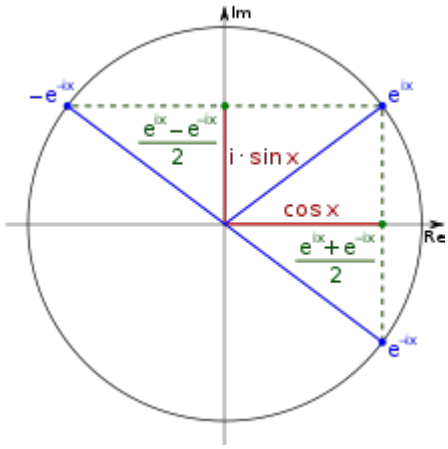
It is worth noting that the Polar Form simplifies the mathematics when used in multiplication or powers of complex numbers

# Relationship to Trigonometry

Euler's Formula provides a connection between Analysis, i.e. relating to limits, and Trigonometry, and provides an interpretation of the Sine and Cosine functions as weighted sums of the Exponential Function

$$\cos x = \Re\left(e^{ix}\right) = \frac{e^{ix} + e^{-ix}}{2}$$

$$\sin x = \Im\left(e^{ix}\right) = \frac{e^{ix} - e^{-ix}}{2}$$




These formulae can even serve as the definition of the Trigonometric Functions for the Complex Argument, x. For example, letting x = iy, we have:

$$\cos(iy) = \frac{e^{-y} + e^{y}}{2} = \cosh(y)$$

$$\sin(iy) = \frac{e^{-y} - e^{y}}{2i} = i \cdot \left(\frac{e^{y} - e^{-y}}{2}\right) = i \cdot \sinh(y)$$

It is worth noting that Complex Exponentials can simplify trigonometry because they are easier to manipulate than their sinusoidal components as we have seen in the Fourier paper:

- ***Differential Equations***: the function $e^{ix}$ is often used to simplify solutions even if the final answer is a real function of sine and cosine.
- ***Signal Processing***: signals that vary periodically are often described as a combination of sinusoidal functions → Fourier Analysis.

# Numerical Solution of an ODE using Euler's Method

The idea behind numerical solutions of a Differential Equation is to replace differentiation by differencing. A computer cannot differentiate a continuous process but it can easily do a difference, a discrete process. Now we introduce the most important tool that will be used in this section, the Taylor Series.

## Taylor Series

*(When a = 0, we have the Maclaurin Series)*

Representation of a function as an infinite sum of terms that are calculated from the values of the function's derivatives ***at a single point***.

- A ***function can be approximated*** by using a finite number of terms of its Taylor Series, producing a Taylor polynomial.
- The Taylor Series of a function is the ***Limit*** of the function's Taylor Polynomials as the degree increases, provided that the Limit exits.
- The Taylor Series of a Real or Complex valued function f(x) that is infinitely differentiable at a real or complex number, a, is the ***Power Series*** :

$$f(a)+\frac{f'(a)}{1!}(x-a)+\frac{f''(a)}{2!}(x-a)^2+\frac{f'''(a)}{3!}(x-a)^3+\ldots$$

This is commonly written as:

$$y(t+\Delta t)=y(t)+y'(t)\Delta t+\frac{1}{2!}y''(t)\Delta t^2+\frac{1}{3!}y'''(t)\Delta t^3+\ldots\frac{1}{n!}y^{(n)}(\tau)\Delta t^n$$

where $\tau$ is some value between $t$ and $\Delta t$

which can also be written

$$\sum_{n=0}^{\infty}\frac{f^{(n)}(a)}{n!}(x-a)^n \qquad \text{or} \qquad \sum_{n=0}^{\infty}\frac{y^{(n)}(t)}{n!}\Delta t^n$$

where $n!$ Denotes factorial n

$f^{(n)}(a)$ denotes the n[th] derivative of f evaluated at a

$y^{(n)}(t)$ denotes the n[th] derivative of y evaluated at t

***Example***: Exponential Function

$$e^x=\sum_{n=0}^{\infty}\frac{x^n}{n!}=1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots$$

***Example***: Binomial Series

$$(1+x)^{\alpha}=\sum_{n=0}^{\infty}\binom{\alpha}{n}x^n \quad \text{where} \quad \binom{\alpha}{n}=\frac{n!}{k!(n-k)!}$$

***Example***: Trig Functions

$$\sin x \;=\; \sum_{n=0}^{\infty}\frac{(-1)^n}{(2n+1)!}x^{2n+1} \;=\; x-\frac{x^3}{3!}+\frac{x^5}{5!}+\ldots \qquad \text{for all x}$$

## Comparison with Fourier Series

Recal from the Fourier paper that the Fourier Series enables one to express a Periodic Function as an infinite sum of Trigonometric Functions:

- ***Local vs.Global***
  - Local: Computation of Taylor Series requires knowledge of the function on an arbitrary small neighbourhood of a point. In Illustration 1 below the Taylor Series of a Sine Wave is based around x = 0 (as a=0, technically, this is a Maclaurian Series). As we add more terms to the approximation the more accurate the approximation becomes further away from our chosen point.
  - Global: Computation of the Fourier Series requires knowing the function on its whole domain interval. The more terms we add to the Fourier Series the more accurate the approximation of the entire function becomes.
- ***Differentiation vs. Integration***
  - Differentiation: Taylor Series is defined for a function which has infinitely many derivatives of a single point, such as a Sine Wave.
  - Integration: The Fourier Series is defined for an Integrable function, such as a Square Wave.
- ***Error***
  - The Taylor Series Error is very small in the neighbourhood of the point where it is computed but large at a distant point, i.e. local error.
  - The Fourier Series Error is distributed along the domain of the function, i.e. global error.

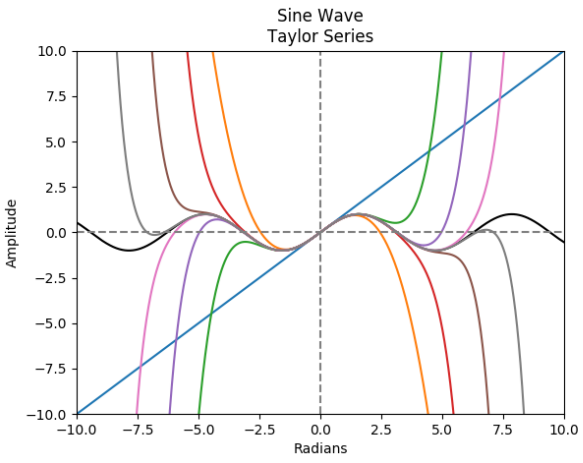
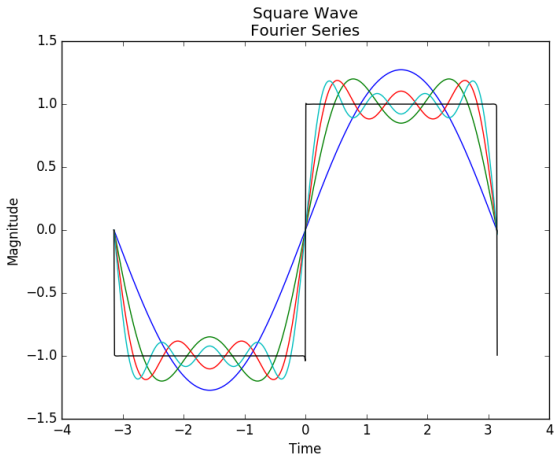
| Taylor Approximation of a Sine Wave | Fourier Approximation of a Square Wave |
|---|---|
|  |  |





*Illustration 1: Taylor Approximation vs. Fourier Approximation*

## Euler's Method

If we truncate the Taylor Series at the first term:

$$y(t+\Delta t)=y(t)+y'(t)\Delta t+\frac{1}{2!}y''(\tau)\Delta t^2$$

We can rearrange this and solve for $y'(t)$ :

$$y'(t)=\frac{y(t+\Delta t)-y(t)}{\Delta t}+O(\Delta t)$$

We can attempt to solve the Differential Equation:

$$\frac{dy(t)}{dt}=f(t,y(t))$$

$$y(0)=y_0$$

by replacing the derivative with a difference:

$$y((n+1)\Delta t)\approx y(n\Delta t)+f(n\Delta t,y(n\Delta t))\Delta t$$

and, starting with $y(0)$ , step forward to solve for time.

If we let $y_n$ be the numerical approximation to the solution, $y(n\Delta t)$ , and $t_n=n\Delta t$ then Euler's Method can be written:

$$y(n+1)=y(n)+f(t_n,y_n)$$

## Runge-Kutta

We improve the accuracy over Euler's method by evaluating the RHS of the differential equation at an intermediate point, the midpoint. The same idea can be applied again, and the function y(n) can be evaluated at more intermediate points, improving the accuracy even more. This is the basis of Runge-Kutta methods, going back to Carl Runge and Martin Kutta. A 2nd Order Runge-Kutta Method can be written out so they don't look messy:

$$k_1=\Delta t f(t_i,y_i)$$

$$k_2=\Delta t f(t_i+\alpha\Delta t,y_i+\beta k_1)$$

$$y_{i+1}=y_i+ak_1+bk_2$$

where we can choose the parameters a, b, α, β so that this method has the highest order Local Truncation Error (LTE).

## Error analysis for time-stepping routines

*Accuracy* and *Stability* are fundamental to numerical analysis and are the key factors in evaluating any numerical integration technique. Therefore, it is essential to evaluate the accuracy and stability of the time-stepping schemes developed. Rarely does it occur that both accuracy and stability work in concert. In fact, they often are off-setting and work directly against each other. Thus, a highly accurate scheme may compromise statbility, whereas a low accuracy scheme may have excellent stability properties.

## Accuracy

It should be clear from our earlier analysis that the Truncation Error (LTE) is $O(\Delta t^2)$, the left out term of the Taylor Series Expansion. Of importance is how this truncation error contributes to the overall error of the numerical solution. Two types of error are important to identify:

***Local Error*** is given by:

$$\epsilon_{k+1} = \boldsymbol{y}(t_{k+1}) - (\boldsymbol{y}(t_k) + \Delta t \cdot \phi)$$

where $\boldsymbol{y}(t_{k+1})$ is the exact solution and
$\boldsymbol{y}(t_k) + \Delta t \cdot \phi$ is a one-step approximation over the time interval $t \in [t_n, t_{n+1}]$

***The Global (cumulative) Error*** is given by

$$E_k = \boldsymbol{y}(t_k) - \boldsymbol{y}_k$$

where $\boldsymbol{y}(t_k)$ is the exact solution and
$\boldsymbol{y}_k$ is the numerical solution

For the Euler Method, we can calculate both the local and global error:

Local: $\epsilon_k \sim O(\Delta t^2)$

Global: $E_k \sim O(\Delta t)$

Thus, the cumulative error is large for the Euler Scheme, i.e. it is not very accurate.

| **Scheme** | **Local Error,** $\epsilon_k$ | **Global Error,** $E_k$ |
|---|---|---|
| Euler | $O(\Delta t^2)$ | $O(\Delta t)$ |
| 2nd Order Runge-Kutta | $O(\Delta t^3)$ | $O(\Delta t^2)$ |
| 4th Order Runge-Kutta | $O(\Delta t^5)$ | $O(\Delta t^4)$ |

*Table 1: Local & Global Discretisation Errors associated with various time-stepping schemes*

Table 1 illustrates various schemes and their associated local and global errors. The error analysis suggests that the error will always decrease in some power of $\Delta t$. Thus, it is tempting to conclude that higher accuracy is easily achieved by taking smaller time steps $\Delta t$. This would be true if it were not for the round-off error in the computer.

## Stability

If a numerical scheme is referred to as ‘stable’ it means that the solution does not blow up to infinity. As a general rule of thumb implicit discretisation schemes tend to more stable than explicit schemes, making them more attractive to practitioners. An example of an implicit scheme can be found in the Fourier paper where we solve the heat equation for a metal rod using a Taylor series expansion and the Crank-Nicolson discretisation method.

# Graph Algorithms

Pairwise connections between items play a critical role in a vast array of computational applications, modelled by abstract mathematical objects called Graphs. Graph theory, initially introduced by Euler, has lead to a number of fundamental graph algorithms that are important in diverse applications. Graphs are fundamantal to a diversity of applications:

- Maps:
  answers quaestions such as ‘what is the shortest route?’
- Web content:
  The web is a graph of referenced (linked) pages making them essentail for Search Engines
- Circuits:
  First introduced as an example in the Fourier paper, solving the current flow around a circuit is key for the electronics industry. There is an additional example of circuits to be found in the Descartes paper which shows the link between linear algebra and graphs.
- Schedules:
  A  key optimisation problem found in logistics and manufacturing. How to get the most from our limited resources and time?
- Commerce:
  Tracking the buying and selling of products in retail and finance gives market insights. Later in this section is an aexample of how Graphs help detect arbitrage opportunities.
- Matching:
  Matching gradutes to jobs or matching couples in dating apps
- Social networks:
  Used haevily in people profiling for political and coommercial campaigns

## Graph Representation

There are two main ways to represent a Graph as a data structure:

1. Adjacency Matrix
   This is equivalent to an Incidence Matrix in Linear Algebra discussed in the Descartes paper. A node-by-node (or vertex-by-vertex in the world of computer science) boolean array is maintained with the entry in row v and column w defined to be true (Incidence Matrices use 1 and 0) if there is an edge adjacent to both node v and node w in the graph, and to be false otherwise. The problem here is that if the number of nodes in the Graph is in the millions using boolean values becomes prohibitive.
2. Array of Adjacency Lists
   A vertex-indexed array of lists of the vertices adjacent to each vertex is maintained. This representation is efficient in terms of space and time.

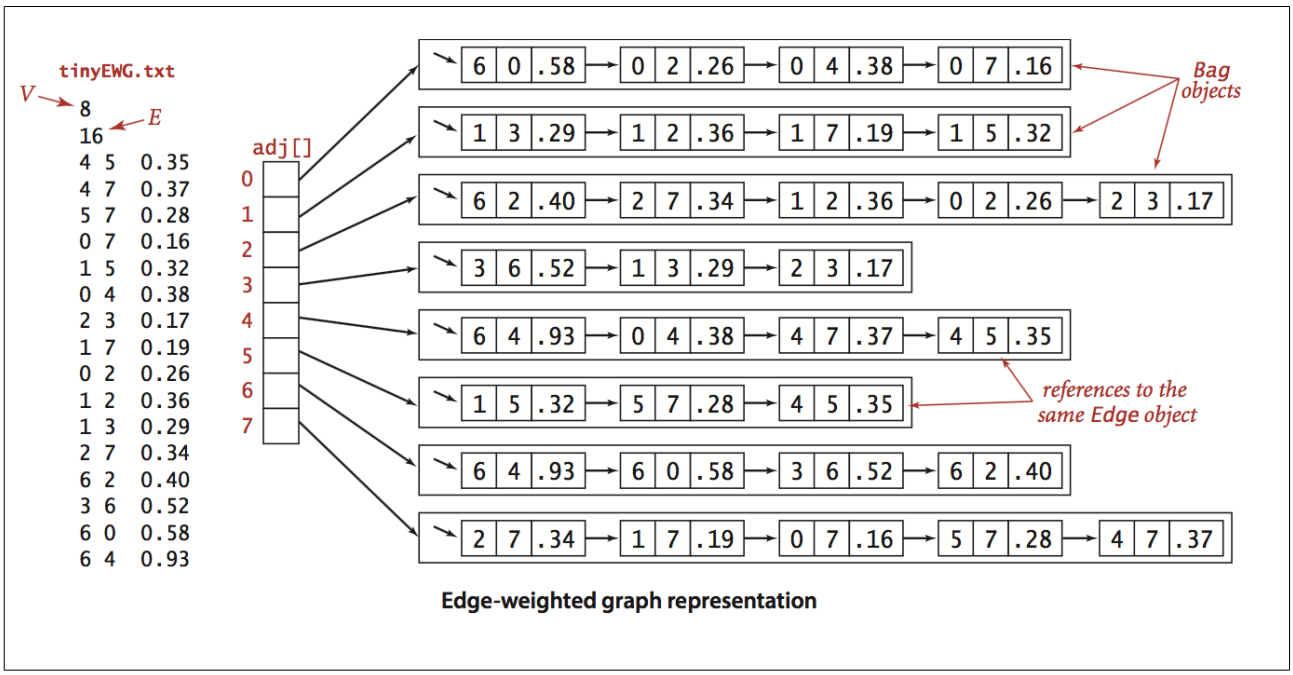



*Illustration 2: Adjacency-lists representation of a Graph*

## Design patterns for graph processing

As we have entered the domain of computer science and software engineering it is important to mention ‘design patterns’. A software design pattern is a reusable solution to a commonly occuring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalised best practices that the programmer can use to solve common problems when designing an application or system.

As with all design patterns the initial design goal is to decouple our implementations from the graph representation. To do so, we develop, for each given task, a task-specific class so that clients can create objects to perform the task. Generally, the constructor does some preprocessing to build data structures so as to efficiently respond to client queries. A typical client program builds a graph, passes that graph to an algorithm implementation class (as argument to a constructor), and then calls client query methods to learn various properties of the graph.

## Minimum Spanning Trees (MST)

From the Descartes paper, recall that a Spanning Tree of a graph is a connected subgraph with no cycles that includes all the vertices. A Minimum Spanning Tree (MST) of an edge-weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree.

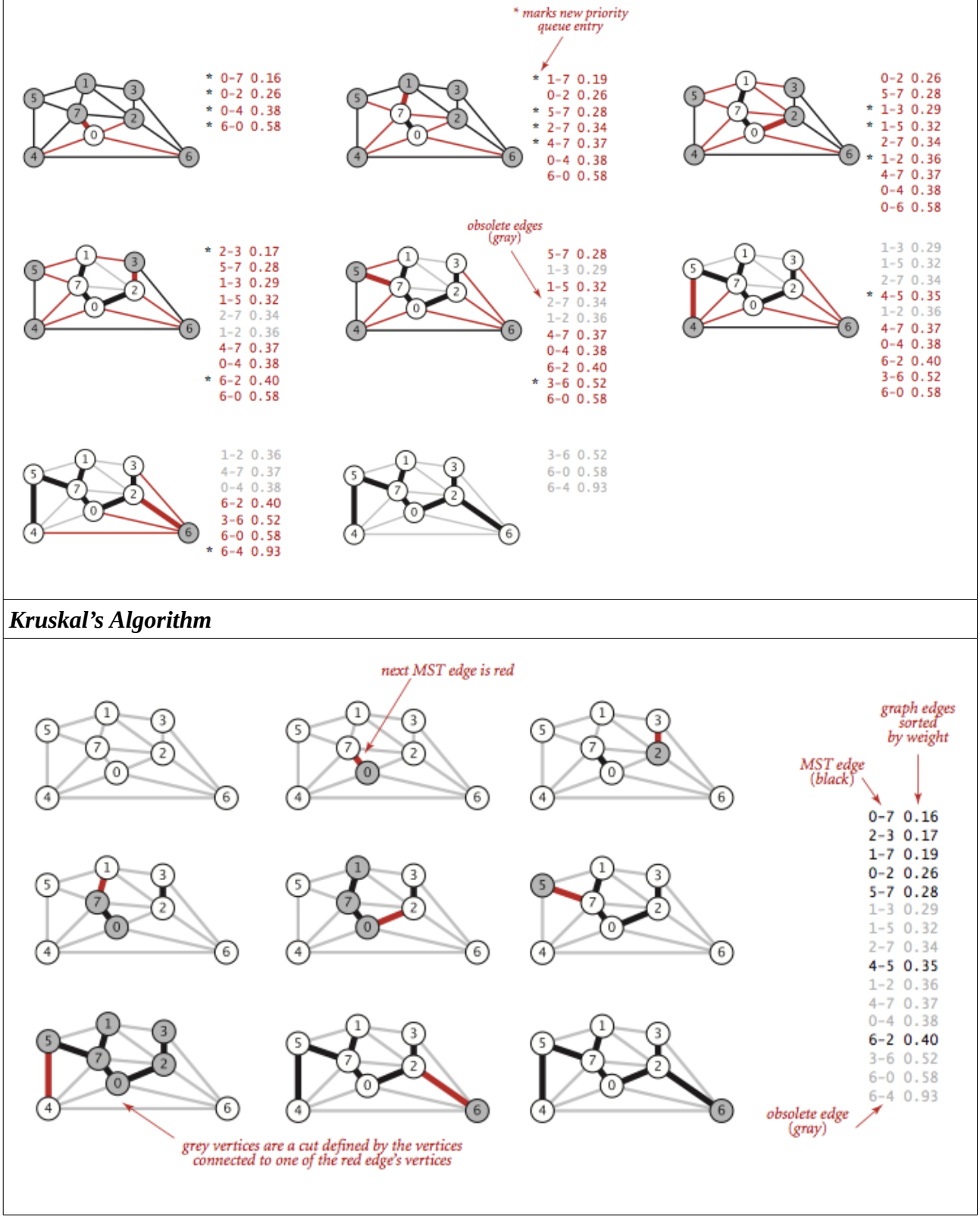



*Illustration 3: Prim's and Kruskal's Algorithms*

We made the assumption that we were looking for a MST of a connected, edge-weighted graph with arbitray (but distinct) weights.

We will look at two classical algorithms for computing MSTs:

1. (Lazy) Prim's Algorithm
    - Attach a new edge to a single growing tree at each step
    - Start with any vertex (node) as a single-vertex tree
    - Then add V-1 edges to it, always taking next the minimum-weight edge that connects a vertex on the tree to a vertex not yet on the tree (a crossing edge for the cut defined by tree vertices).
2. Kruskal's Algorithm
    - Process the edges in order of their weigh values (smallest to largest)
    - Taking for the MST each edge that does not form a cycle with edges previously added
    - Stopping after adding V-1 edges have been taken

## Performance Characteristics

Prim's algorithm builds the MST one edge at a time, finding a new edge to attach to a single growing tree at each step. Krustal's algorithm also builds the MST one edge at a time; but, by contrast, it finds an edge that connects two trees in a forest of growing trees. We start with adegenerate forest of V single-vertex trees and perform the operation of combining two trees (using the shortest edge possible) until there is just one tree left; the MST.

| ***Algorithm*** | ***Space*** | ***Time*** |
|---|---|---|
| ***Minimum Spanning Tree*** | | |
| Lazy Prim | E | E log E |
| Eager Prim | V | E log V |
| Krustal | E | E log E |
| Fredman-Tarjan | V | E + V log V |
| Chazelle | V | Close but not quite E |
| Impossible? | V | E? |
| ***Shortest Path*** | | |
| Dijkstra | V | E log V |
| Bellman-Ford | V | E + V |

*Table 2: Performance characteristics of MST algorithms*
*(worst-case order of growth for V vertices and E edges)*

We only discussed two MST algorithms: Lazy Prim and Krustal, but, as the MST problem is a heavily studied problem it has a large body of research dedicated to it. Table 2 is a list of the performance characteristics of a number of MSP algorithms. E is the number of Edges and V is the number of verices (or nodes). For completeness, Shortest Path algorithms have also been included.

## Shortest Path

Probably the most intuitive graph-processing problem is to get directions from one place to another. The nodes, or vertices, correspond to intersections and edges correspond to roads, with weights on the edges that model the cost, perhaps distance or travel time. In this model, the problem is easy to formulates:

*Find the lowest-cost way to get from one vertex (node) to another*

### Data Structure for shortest paths

The data structures that we need to represent shortest paths are straight forward.:

- Edges on the shortest-path tree:
  We use a parent-edge representation in the form of a vertex-indexed **edgeTo[]** of DirectedEdges objects, where **edgeTo[v]** is the edge that connects v to its parent in the tree (the last edge on a shortest path from s to v).
- Distance to the source:
  We use a vertex-indexed array **distTo[]** such that **distTo[v]** is the length of the shortest known path from s to v.

### Edge Relaxation

Our shortest-path implementations are based on a simple operation known as relaxation. We start knowing only the graph's edges and weights, with the **distTo[]** entry for the source initialised to 0 and all of the other **distTo[]** entries initialised to infinity. As an algorithm proceeds, it gathers information about the shortest paths that connect the source to each vertex encountered in our **edgeTo[]** and **distTo[]** data structures. By updating this information when we encounter edges, we can make new inferences about shortest paths.

### Shortest Path Algorithms

We will look at two classical algorithms for computing Shortest Paths:

1. Dijkstra:
   - As discussed above, Prim's algorithm for finding the MST of an edge-weighted graph: we build the MST by attaching a new edge to a single growing tree at each step. Dijkstra's algorithm is an analogous scheme to compute the Shortest Path.
   - We begin by initialising **distTo[s]** to 0 and all other **distTo[]** entries to +ive infinity
   - Then we relax and add to the tree a non-tree vertex with the lowest **distTo[]** value
   - Contining until all vertices are on the tree ***or*** no non-tree vertex has a finite **distTo[]** value.
2. Bellman-Ford:
   - Bellman-Ford is slower than Dijkstra in computing shortest paths from a single source to all the other vertices in  aweighted digraph ***but***

- It is capable of handling graphs in which some of the edge weights are -ive numbers.
- If a graph conatins a '***negative-cycle***' that is reachable from the source, then there is no cheapest path:
  - Any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle.
- In such a case, the Bellman-Ford algorithm can detect cycles and report their existence

```
import networkx as nx
import numpy as np

def bellman_ford(G, source, currDict):
    '''
    This implementation takes in a graph and fills
    two lists (dist and pred) about the
    shortest path from the source to each node/vertex
    '''

    # Step 0: Create graph, G

    # Step 1: initialize graph
    dist = []
    pred = []
    for v in list(G.nodes()):
        dist.append(np.inf) # Initialize the distance to all vertices to infinity
        pred.append(None)   # And having a null predecessor

    dist[source] = 0        # The distance from the source to itself is zero

    # Step 2: relax edges repeatedly
    weights = nx.get_edge_attributes(G, 'weight')
    for i in np.arange(1, len(list(G.nodes))):
        for edge in G.edges:
            u = edge[0]; v = edge[1]; w = weights[(u, v)]
            if dist[u] + w < dist[v]:
                dist[v] =  dist[u] + w
                pred[v] = u

    # Step 3: check for negative-weight cycles
    for edge in G.edges:
        u = edge[0]; v = edge[1]; w = weights[(u, v)]
        if dist[u] + w < dist[v]:
            print("Graph contains a negative-weight cycle")
```

*Illustration 4: Implementation of Bellman-Ford algorithm in Python*

Illustration 4 above is an implementation of the Bellman-Ford algorithm in Python which was used in the Arbitrage application that follows.

Performance characteristics of Shortest path algorithms, together with Minimum Spanning Tree algorithms, can be found in Table 2.

## Application: Arbitrage in currency exchange

We can reduce the arbitrage problem to a *negative-cycle-detection* problem in edge weighted digraphs using the Bellman-Ford algorithm.

Consider a market for financial transactions that is based on trading curriencies. Our example only includes 5 of the hundreds of currencies that are traded on modern markets; US dollars (USD), Euros (EUR), British pounds (GBP), Swiss francs, and Canadian dollars (CAD).

| | USD | EUR | GBP | CHF | CAD |
|---|---|---|---|---|---|
| **USD** | **1.0** | 0.741 | 0.657 | 1.061 | 1.005 |
| **EUR** | 1.349 | **1.0** | 0.888 | 1.433 | 1.366 |
| **GBP** | 1.521 | 1.126 | **1.0** | 1.614 | 1.538 |
| **CHF** | 0.942 | 0.698 | 0.619 | **1.0** | 0.953 |
| **CAD** | 0.995 | 0.732 | 0.650 | 1.049 | **1.0** |

*Table 3: Conversion rates between 5 currencies*

As shown in Table 3 the tth number on line s represents a conversion rate.i.e. the number of units of the currency named on row s that is needed to buy 1 unit of the currency named in row t. This table is equivalent to a complete edge-weighted digraph with a vertex (node) corresponding to each currency and an edge corresponding to each conversion rate. An edge **s→t** with weght **x** corresponds to a conversion from **s** to **t** at exchange rate **x**. paths in the digraph specify multistep conversions.

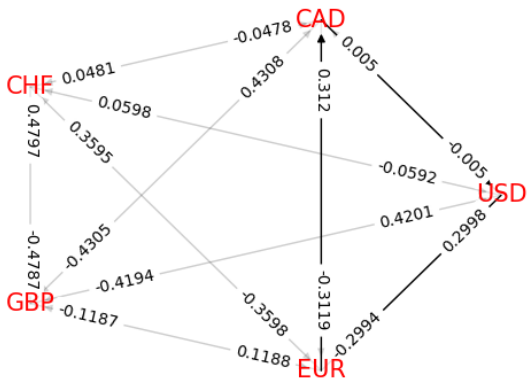
| ***Negative Cycle*** | ***Results of Bellman-Ford*** |
|---|---|
|  | ***1st Transaction:***<br>1000.0 USD = 741.0 EUR<br>***2nd Transaction:***<br>741.0 EUR = 1012.206 CAD<br>***3rd Tranaction:***<br>1012.206 CAD = 1007.14497 USD<br><br>***Arbitrage Profit:***<br>US $7.15 |

*Illustration 5: An Arbitrage Opportunity*

What is of interest is the case, shown in Illustration 5, where the product of the edge weights is smaller than the weight of the edge from the last vertex (node) to the first. If we convert US $1000 to EUR at  arate of 0.741 Euros per US$ we get 741 Euros. If we then convert our Eruros to Canadian dolars at a ratr of 1.366 we ger CAN $1012.206. If we convert our Canadian $1012.206 back to US $, we get $1007.15, a US $7.15 profit, purely from arbitrage.

# Machine Learning Optimisation Algorithms

## Supervised Learning: Gradient Descent with Back-Propagation

One of the most successful approaches to supervised machine learning can be called "numerical" or ***gradient-based learning***. The learning machine computes a ***forward pass*** function:

$$Y^p = F(Z^p, W)$$

where $\mathbf{Z^p}$ is the p-th input pattern and
**W** represents the collection of adjustable parameters in the system

In a pattern recognition setting, the output $Y^p$ may be interpreted as the recognised class label of pattern $Z^p$, or as scores or probabilities associated with each class.

A ***loss function***:

$$E^p = D(D^p, F(W, Z^p))$$

measures the discrepancy between:

- $D^p$, the "correct" or desired output, i.e. label, for pattern $Z^p$, and
- the output produced by the system.

The ***average loss function*** $\boldsymbol{E_{train}(W)}$ is the average of the errors $E^p$ over a set of labeled examples called the training set $\{(Z^1, D^1), \ldots.(Z^p, D^p)\}$.

In the simplest setting, the learning problem consists in finding the value W that ***minimises*** $\boldsymbol{E_{train}(W)}$. In practice, the performance is estimated by measuring the accuracy on a set of samples disjoint from the training set, called the test set.

The gap between the expected error rate on the test set $\boldsymbol{E_{test}}$ and the error rate on the training set $\boldsymbol{E_{train}}$ decreases with the number of training samples.

## Gradient-Based Learning

Gradient-Based Learning draws on the fact that it is generally much easier to minimise a resonably smooth, continuous function than a discrete (combinatorial) function. The loss function can be minimised by estimating the ***impact of small variations of the parameter values on the loss function***. This is measured by the gradient of the loss function w.r.t. the parameters.

Efficient learning algorithms can be devised when the gradient vector can be computed ***analytically*** (as opposed to numerically through perturbations). This is the basis of numerous gradient-based learning algorithms with continuous-valued parameters. In the procedures described here, the set of parameters ***W*** is a real-valued vector wrt ***E(W)*** which is continuous as well as differentiable almost everywhere.

The ***simplest minimisation procedure*** in a such a setting is the ***gradient descent algorithm*** where ***W*** is iteratively adjusted, i.e. the network weights are updated, as follows:

$$W_k = W_{k-1} - \epsilon \left( \frac{\partial E(W)}{\partial W} \right)$$

In the simplest case, ε is a scalar constant. More sophisticated procedures use variable ε, although their usefulness to large learning machines is very limited.

## Stochastic Gradient Descent

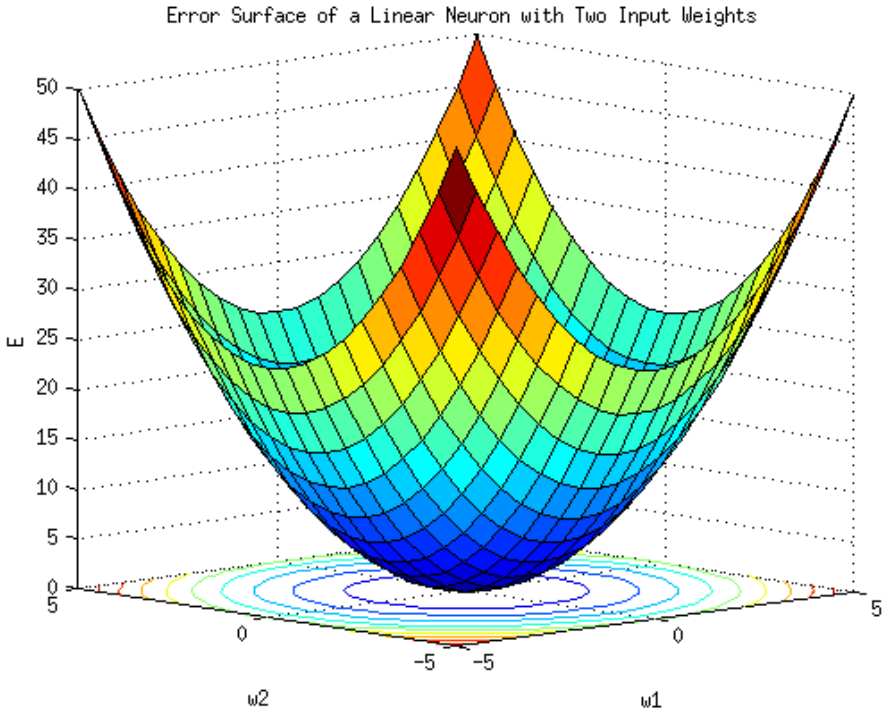



*Illustration 6: Error Surface of a Linear Neuron with 2 Input Weights, w1 & w2*

A popular minimisation procedure is the ***stochastic gradient algorithm***, also called the on-line update. It consists in updating the parameter vector using a noisy, or approximated, version of the average gradient. In the most common instance of it, W is updated on the basis of a single sample:

$$W_k = W_{k-1} - \epsilon \left( \frac{\partial E^{pk}(W)}{\partial W} \right)$$

With this procedure the parameter vector fluctuates around an average trajectory, but usually converges considerably faster than regular gradient descent on large training sets with redundant samples (such as those found in speech or character redognition).

We now need to compute the gradient, $\partial E^{pk}(W) / \partial W$ and for that we use backward propagation.

## Gradient Back-Propagation

The basic idea of back-propagation is that gradients can be computed efficiently ***by propagation from the output to the input***. As an aside, it appears odd that ***local minima*** do not seem to be a problem for multi-layer neural networks is somewhat of a theoretical mystery, but it appears that a

network that is over-sized for a task, the presence of extra dimensions in parameter space ***reduces the risk of unattainable regions***.

Back-propagation is by far the most widely used neural-network learning algorithm, and probably the most widely used learning algorithm of any form.

The back-propagation procedure is used to compute the gradients of the loss function w.r.t. ***all the parameters in the system***. For example, let us consider a system built as a cascade of modules, each of which implements a function, i.e. a layer computation:

$$X_n = F_n(W_n, X_{n-1})$$

where $\mathbf{X_n}$ is a vector representing the output of the module
$\mathbf{W_n}$ is the vector of tunable parameters in the module (a subset of W)
$\mathbf{X_{n-1}}$ is the module's input vector (as well as the previous module's output vector).

The input $X_0$ to the first module is the input pattern $Z^p$.

If the partial derivative of $E^p$ w.r.t. $X_n$ is known, then the partial derivatives of $E^p$ w.r.t. $W_n$ and $X_{n-1}$ can be computed using the backward recurrence:

$$\frac{\partial E^p}{\partial W_n} = \frac{\partial F(W_n, X_{n-1})}{\partial W} \cdot \frac{\partial E^p}{\partial X_n} \quad \text{and} \quad \frac{\partial E^p}{\partial X_{n-1}} = \frac{\partial F(W_n, X_{n-1})}{\partial X} \cdot \frac{\partial E^p}{\partial X_n}$$

where $\frac{\partial \boldsymbol{F}(\boldsymbol{W_n}, \boldsymbol{X_{n-1}})}{\partial \boldsymbol{W}}$ is the Jacobian of F wrt W evaluated at the point ($W_n$, $X_{n-1}$) and

$\frac{\partial \boldsymbol{F}(\boldsymbol{W_n}, \boldsymbol{X_{n-1}})}{\partial \boldsymbol{X}}$ is the Jacobian of F wrt X.

As we know, the ***Jacobian*** of a vector function is a matrix containing the partial derivatives of all the ouputs wrt all the inputs.

- The LHS equation computes some terms of the gradient of $E^p(W)$.
- The RHS equation generates a backward recurrence, as in the well-known back-propagation procedure for neural networks.
- We can average the gradients over the training patterns to obtain the full gradient.

Traditional multi-layer neural networks are a special case of the above where the state information $X_n$ is represented with fixed-sized vectors, and where the modules are alternated layers of matrix multiplications (the weights) and component-wise sigmoid functions (the neurons).

However, the state information in complex recognition systems is best represented by ***graphs*** with numerical information attached to the arcs. In this case, each module, called a Graph Transformer, takes one or more graphs as input, and produces a graph as output. This means that Gradient-Based Learning can be used to train all the parameters in all the modules so as to ***minimise a global loss function***.

Pseudocode for a stochastic gradient descent optimisation algorithm:

initialise ***model*** weights (often small random values)

**do**

**forEach** training example named input

prediction = neural_net_output(W, input) # ***forward pass***

actual = label(input) # ***label***

compute error (prediction – actual) # ***loss function***

compute gradients, $\frac{\partial E^{pn}(W)}{\partial W}$ # ***backward propagation***

$W_n = W_{n-1} - \epsilon \cdot \frac{\partial E^{pn}(W)}{\partial W}$ # ***update network weights***

**until** all input examples classified properly or other stopping criterion satisfied

**return** trained model

ArtNet is an example of a Neural Network learning algorithm using Back-propagation.

## Unsupervised Learning: Expectation-Maximisation (EM)

The Expectation-Maximisation algorithm is frequently used for data clustering in machine learning, as seen in the exampe in Illustration 7 and computer vision; natural language processing; unsupervised induction of probabilistic context-free grammars; parameter estimation of mixed models amongst other examples in finance, engineering and science.

The goal of the EM algorithm is to find M***aximum Likelihood solutions*** for models having ***latent variables***. We denote the set of all observed data by **X** and similarly we denote the set of all latent variables by **Z**. The set of all model parameters is denoted by $\boldsymbol{\theta}$, and so the ***log likelihood function*** is given by

$$\ln p(\boldsymbol{X}|\boldsymbol{\theta}) = \ln\{\sum_{\boldsymbol{Z}} p(\boldsymbol{X},\boldsymbol{Z}|\boldsymbol{\theta})\} \quad .$$

As we are rarely given the complete data set {**X**, **Z**} we tend to only have the ***incomplete*** data **X**. Our state of knowledge of the values of the latent variables in **Z** is given by the ***posterior distribution*** $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$.

Because we cannot use the complete-data log likelihood, we consider instead its expected value under the posterior distribution of the latent variable, which corresponds to the ***E-step*** of the EM algorithm. In the subsequent ***M-step***, we maximise this expectation. If the current estimate for the parameters is denoted as $\boldsymbol{\theta}^{old}$ , then a pair of successive E and M steps gives rise to a revised estimate $\boldsymbol{\theta}^{new}$ . The algorithm is initialised by choosing some starting value for the parameters $\boldsymbol{\theta}_0$ .

***E-Step:***

We use the current parameter $\boldsymbol{\theta}^{old}$ to find the posterior distribution of the latent variables given by $p(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{\theta}^{old})$ . We then use this posterior distribution to find the expectation of the complete-data log likelihood evaluated for some general parameter value **θ**. This expectation, denoted $Q(\boldsymbol{\theta},\boldsymbol{\theta}^{old})$ is given by

$$Q(\boldsymbol{\theta},\boldsymbol{\theta}^{old}) = \sum_{\boldsymbol{Z}} p(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{\theta}^{old}) \ln p(\boldsymbol{X},\boldsymbol{Z}|\boldsymbol{\theta}) \quad .$$

***M-step***:

We determine the revised parameter estimate $\boldsymbol{\theta}^{new}$ by maximising this function

$$\boldsymbol{\theta}^{new} = \underset{\boldsymbol{\theta}}{arg\,max}\, Q(\boldsymbol{\theta},\boldsymbol{\theta}^{old}) \quad .$$

The general EM algorithm is summarised below. It has the property that each cycle of EM will increase the incomplete-data log likelihood, unless, of course, it is already at a local maximum.

## The General EM Algorithm

Given a joint distribution $p(\boldsymbol{X},\boldsymbol{Z}|\boldsymbol{\theta})$ over observed variable **X** and latent variables **Z**, governed by parameters $\boldsymbol{\theta}$, the goal is to maximise the likelihood function $p(\boldsymbol{X}|\boldsymbol{\theta})$ wrt $\boldsymbol{\theta}$.

1. Choose an initial setting for the parameters $\boldsymbol{\theta}^{old}$
2. E-Step: Evaluate $p(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{\theta}^{old})$
3. M-Step: Evalute $\boldsymbol{\theta}^{new}$ given by:

$$\boldsymbol{\theta}^{new} = \underset{\boldsymbol{\theta}}{arg\,max}\, Q(\boldsymbol{\theta},\boldsymbol{\theta}^{old})$$

where

$$Q(\boldsymbol{\theta},\boldsymbol{\theta}^{old}) = \sum_{\boldsymbol{Z}} p(\boldsymbol{Z}|\boldsymbol{X},\boldsymbol{\theta}^{old}) \ln p(\boldsymbol{X},\boldsymbol{Z}|\boldsymbol{\theta})$$

4. Check for convergence of either the log likelihood or the parameter values.
   If the convergence criterion is not satisfied, then let

$$\boldsymbol{\theta}^{old} \leftarrow \boldsymbol{\theta}^{new}$$

and return to step 2

The EM algorithm can also be used to find MAP (maximum posterior) solutions for models in which a prior $p(\boldsymbol{\theta})$ is defined over the parameters. In this case the E-step remains the same as in the maximum likelihood case, whereas in the M-step the quantity to be maximised is given by:

$$Q(\boldsymbol{\theta},\boldsymbol{\theta}^{new}) + \ln p(\boldsymbol{\theta})$$

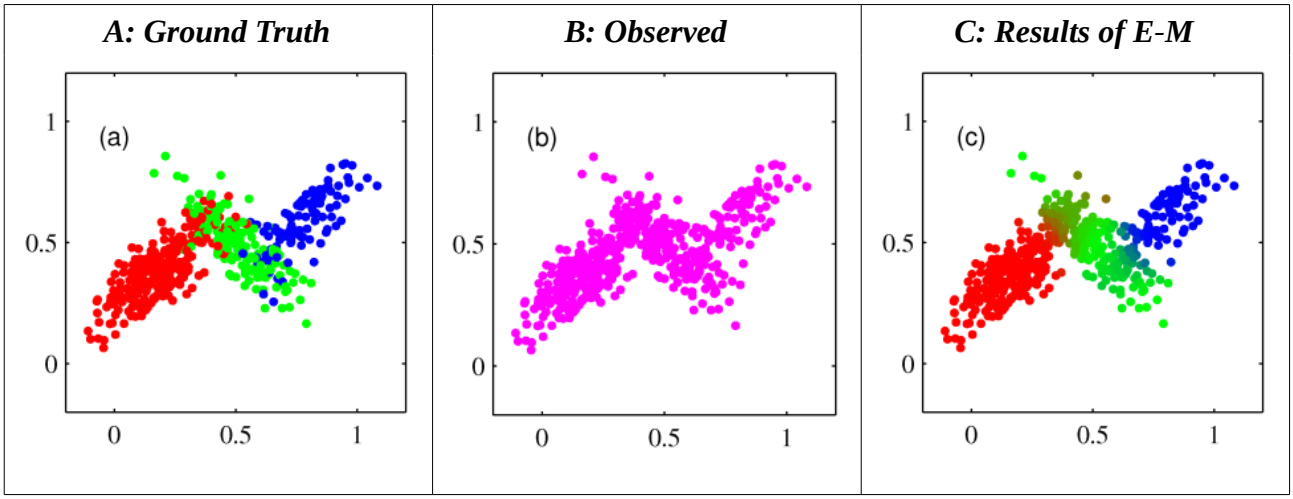


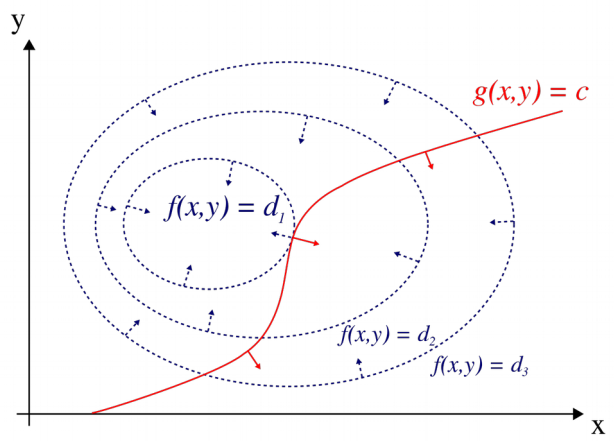

*Illustration 7: EM algorithm performed on mixture of Gaussian data, **X**.*

In Illustration 7, ***A*** is the ground truth data, where the colors indicate which Gaussian gave rise to that data point. ***B*** depicts what we observe. It becomes clear that we are not given the Gaussian to which each data point belongs to and so this information is hidden. Assume all we know is there are 3 Gaussians, i.e. the three states of **Z**, that generated the data. ***C*** shows the results of running EM after some iterations on the data, that is the EM algorithm has labeled each point by a distribution of how likely they are to be from each Gaussian. You can see that points near other Gaussians are colored as a mixture.

# Lagrange Multipliers

Strategy for finding local Maxima and Minima of a function subject to Equality Constraints. It is widely used in Economics and the Modern Portfolio Theory (MPT).

## Intuition




- At a maximum, f(x,y) cannot be increasing in the direction of any neighbouring point where g=0.
- If it were, we could walk along g=0 to get higher, meaning that the starting point wasn't actually the maximum.
- We want points (x,y) where g(x,y)=0 and $\nabla_{x,y} f = \lambda \nabla_{x,y} g$ for some λ.
- λ is required because, although the two gradient vectors are parallel, the magnitudes of the gradient vectors are generally not equal.
- To incorporate these conditions into one equation, we introduce an auxillary function: $L(x,y,\lambda) = f(x,y) - \lambda \cdot g(x,y)$ and solve $\nabla_{x,y,\lambda} L(x,y,\lambda) = 0$ , as below.

*Table 4: The red curve shows the constraint g(x, y) = c. The blue curves are contours of f(x, y). The point where the red constraint tangentially touches a blue contour is the maximum of f(x, y) along the constraint, since d1 > d2.*

## Single Constraint

- One constraint and only two variables
- functions, f and g, have continuous first partial derivatives

Maximise f(x, y)
Subject to g(x, y) = 0

- We introduce a new variable, λ, called a Lagrange Multiplier
- Then study the Lagrange Function defined by:

$$L(x,y,\lambda) = f(x,y) - \lambda \cdot g(x,y)$$

and solve

$$\nabla_{x,y,\lambda} L(x,y,\lambda) = 0$$

which is equivalent to 3 equations with 3 unknowns.

The method generalises readily to functions on n variables:

$$\nabla_{x_1,\ldots,x_n,\lambda} L(x_1,\ldots,x_n,\lambda) = 0$$

which amounts to solving n+1 equations with n+1 unknowns

## Example:

Maximise f(x, y) = x + y
Subject to $x^2+y^2=1$

***Step 1***: The Lagrange Function is:

$$L(x,y,\lambda)=f(x,y)-\lambda\cdot g(x,y)$$

after substuiting the equations, we get:

$$L(x,y,\lambda)=x+y+\lambda\cdot(x^2+y^2-1)$$

***Step 2***: calculate the Gradients

$$\nabla_{x,y,\lambda}L(x,y,\lambda)=(\frac{\partial L}{\partial x},\frac{\partial L}{\partial y},\frac{\partial L}{\partial \lambda})$$

which gives

$$\nabla_{x,y,\lambda}L(x,y,\lambda)=(1+2x\lambda,1+2y\lambda,x^2+y^2-1)$$

and therefore gives us three equations to find three variables:

$$\nabla_{x,y,\lambda}L(x,y,\lambda)=0 \quad \Leftrightarrow \quad \begin{aligned} 1+2x\lambda&=0\\ 1+2y\lambda&=0\\ x^2+y^2-1&=0\end{aligned}$$

***Step 3***: The first two equations yield:

$$x=y=-\frac{1}{2\lambda} \qquad \lambda\neq 0$$

By substuiting into the last equation we have:

$$\frac{1}{4\lambda^2}+\frac{1}{4\lambda^2}-1=0 \quad \text{so} \quad \lambda=\pm\frac{1}{\sqrt{(2)}}$$

***Step 4***: Which implies that the stationary points of $L$ are:

$$(\frac{\sqrt{(2)}}{2},\frac{\sqrt{(2)}}{2,}-\frac{1}{\sqrt{(2)}}) \ , \qquad (-\frac{\sqrt{(2)}}{2},-\frac{\sqrt{(2)}}{2},\frac{1}{\sqrt{(2)}})$$

***Step 5***: Evaluating the Objective Function, f, at these points yields:

$$f(\frac{\sqrt{(2)}}{2},\frac{\sqrt{(2)}}{2})=\sqrt{(2)} \qquad f(-\frac{\sqrt{(2)}}{2},-\frac{\sqrt{(2)}}{2})=-\sqrt{(2)}$$

Thus the ***Constrained Maximum*** is $\sqrt{(2)}$
And the ***Constrained Minimum*** is $-\sqrt{(2)}$

# Example: Modern Portfolio Theory (MPT)

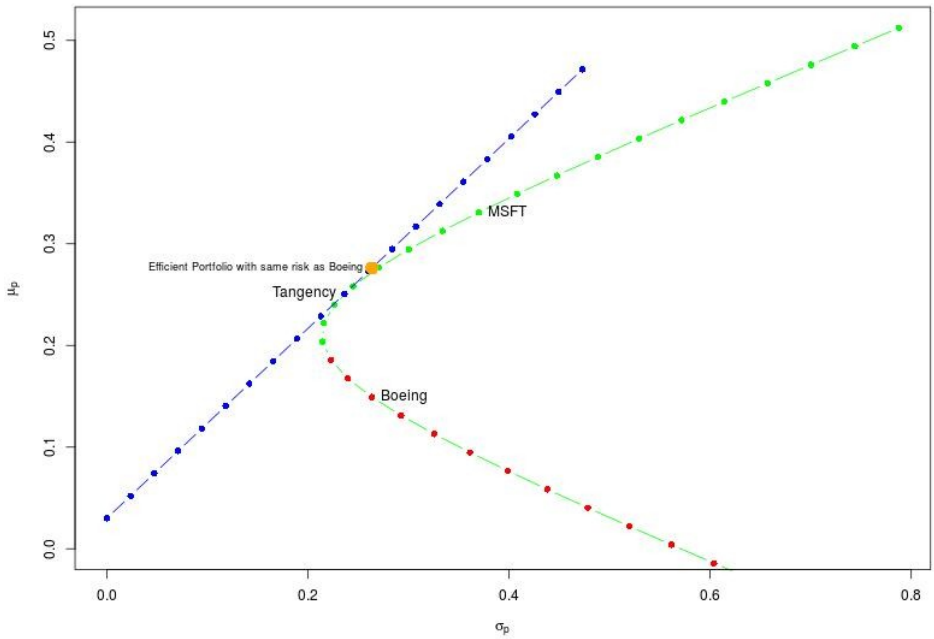



*Illustration 8: Efficient Portfolio with the Risk of Boeing Stock*

Modern Portfolio Theory (MPT), or mean-variance analysis, is a mathematical framework for assembling a portfolio of assets such for which the expected ***return is maximised*** for a given level of risk. It is a formalisation and extension of diverisification in investing, the idea that owning different kinds of financial assets is less risky than owning only one type. Its key insight is that an asset's risk and return should not be assessed by itself, but by how it contributes to a portfolio's overall risk and return. It uses variance of asset prices as a proxy for risk. Economist Harry Markowitz introduced MPT in a 1952 essay, for which he was later awarded a Nobel Prize in Economics.

There is a trade-off between risk and reward and depends on an investor's risk aversion characteristic. The implication is that investors will pursue an investment portfolio that maximes their return for a stated level of risk. Markowitz noted the variance can be limited by diversifying a portfolio which can reduce exposure to an individual asset and may allow the same portfolio return with reduced risk.

Illustration 8 shows the expected return versus the standard deviation. The left boundary is a hyperbola and the upper edge of this region is the ***efficient frontier*** in the absence of a risk-free asset – the Markowitz bullet. Combinations along this upper edge represent portfolios for which there is lowest risk for a given level of expected return. The tangent to the hyperbola at the tangency point indicates the best possible allocation line of ***Tangency Portfolios***. Through the application of the Modern Portfolio Theory, we have an efficient portfolio that has a higher expected return than holding only Boeing stock but with the same expected risk. The risk and reward trade-off can now easily be managed by moving along the tangent line. To the left, we become more risk averse but to the right we have a better chance of increased returns.

# Discrete Optimisation

## Max flow – min cut Theorem:

The maximal flow in a network equals the total capacity across the minimal cut.

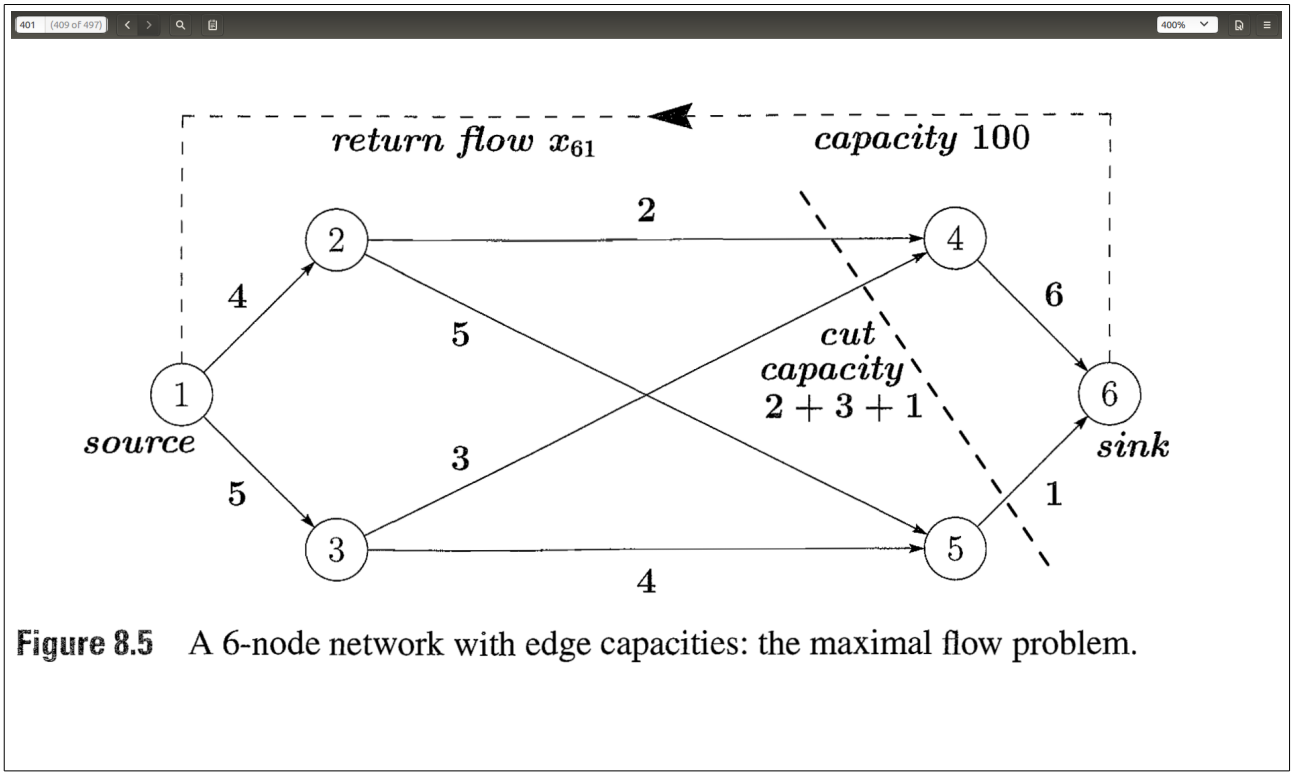



**Figure 8.5** A 6-node network with edge capacities: the maximal flow problem.

*Illustration 9: A 6-node network with edge capacities: the maximal flow problem*

***Constraints:***

- Flows are non-negative: $x_{ij} >= 0$ going with the arrows
- Flows cannot exceed the capacities marked on the edges: $x_{ij} <= c_{ij}$
- Directions given by the arrows cannot be reversed
- Kirchoff Current Law (KCL): flow into each node equals the flow out: $Ax = 0$
  - A is the node-edge incidence matrix
  - A has a row for every node
  - A has a +1, -1 column for every edge
- Flow on the two edges into the sink cannot exceed 6 + 1 = 7

***Unknowns:***

- Flows $x_{ij}$ from node *i* to node *j*

By maximising the return flow $x_{61}$, in Illustration 9, we maximise the total flow into the sink:

**Maximal Flow:** **Maximise $x_{61}$ subject to $Ax = 0$ and $0 <= x_{ij} <= c_{ij}$**

The key is to find a cut in the network across which all capacities are filled. That cut separates nodes 5 and 6 from the others. The edges that go forward across the cut have total capacity 6.

- ***Weak Duality***: every cut gives a bound to the total flow
- ***Full Duality***: cut of smallest capacity (the minimal cut) is filled by the maximal flow

A 'cut' splits the nodes into two groups ***S (source)*** and ***T (sink)***. Its capacity is the sum of the capacities of all edges crossing the cut (from S to T). Several cuts might have the same capacity. The total flow can never be greater than the total capacity across the minimal cut. The problem here and in all ***duality***, is to show that equality is achieved by the right flow and the right cut.

### Ford-Fulkerson Algorithm

The Ford-Fulkerson Algorithm is an effective way to solve max-flow problems. It is a generic method for increasing flows incrementally along paths from source to sink. A more descriptive term is ***augmenting-path algorithm***:

- Start with zero flow everywhere
- Increase the flow along any augmenting path from source to sink
  - with no full forward edges or empty backward edges
- Continue until there are no such paths in the network

## Reduction

A problem A reduces to another problem B if we can use an algorithm that solves B to develop an algorithm that solves A. Together with sorting, shortest paths and maxflow are important problem-solving methods and a number of previously mentioned problems can be reduced to them:

### Shortest-path reductions

- Parallel precedence-constrained scheduling:
  - Given a set of jobs with precedence constraints
  - Schedule jobs on identical processors such that they are completed in the minimum amount of time while respecting the constraints?
- Arbitrage:
  - Find an arbitrage opportunity in a given table of currency conversion rates

### Maxflow reductions

- Job placement
  - Is it possible to match every student with a job?
  - What is the maximum number of jobs that can be filled?
- Product distribution
  - Is it possible to get product from warehouses to retail outlets such that supply meets demand?
- Network reliability
  - What is the minimum number of trunk lines that can be cut to disconnect some pair of computers?

## Linear Programming

One of the cornerstones of operations research is linear programming. It refers to the idea of reducing a given problem to the following mathematical formulation.

Given a set of:

- M linear inequalities and
- Linear equations involving N variables, and a
- Linear objective function of the N variables

Find an assignment of values to the variables that maximises the objective function, or report that no feasible assignment exists.

This topic is discussed further in the Descartes paper.

# Theory of Intractability

The purpose of intractability is to seperate problems that can be solved in *polynomial time* from problems that (probably) require *exponential time ($2^N$ at least)*, to solve in the worst case.

## Search problems

A search problem is a problem having solutions with the property that the time needed to certify that any solution is correct is bounded by a polynomial in the size of the input. We say that an algorithm solves a search problem if, given any input, it either produces a solution or reports that none exists.

## Definitions

**NP** is the set of all search problems we ***aspire to solve*** feasibly

A search problem A is said to be ***NP****-complete* (**NPC**) if all problems in NP poly-time reduce to A

**P** is the set of all search problems that ***can be solved in polynomial time***

Although it has yet to be proven, it is suspected that

**P != NP**

## Classifying problems

Classifying problems as being easy to solve (in **P**) or hard to solve (**NP**-complete) can be:

- Straight forward
  - Gaussian elimination proves that linear equation satisfiability is in **P**
- Tricky but not difficult
  - Event-based simulation of N colliding particles is in **P**
- Extremely challenging
  - Linear programming is in **P**
- Open
  - Factor (given an integer, find a non-trivial factor) is still unclassified.
  - Travelling Salesman

# Simulated Annealing – a return to Monte Carlo

In this section we apply a Monte Carlo method called simulated annealing, an important tool for solving difficult optimisation problems, mostly without any relation to physics. In this approach, a discrete or continuous optimisation problem is mapped onto an artificial physical system whose ground state (at zero temperature or infinite pressure) contains the solution to the original task.

In simulated annealing, the equivalent of temperature is a measure of the randomness by which changes are made to the path, seeking to minimise it. When the temperature is high, larger random changes are made, avoiding the risk of becoming trapped in a local minimum (of which there are usually many in a typical travelling salesman problem), then honing in on a near-optimal minimum as the temperature falls. The temperature falls in a series of steps on an exponential decay schedule.

## Solution to the Travelling Salesman NP-complete Problem

The travelling salesman problem is a problem in graph theory requiring the most efficient, i.e. least total distance, Hamiltonian cycle a saleman can take through each of N cities. No general method of the solution is known, and the problem is NP-complete. That said, it can be solved using Simulated Annealing, as shown in Illustration 10 below.

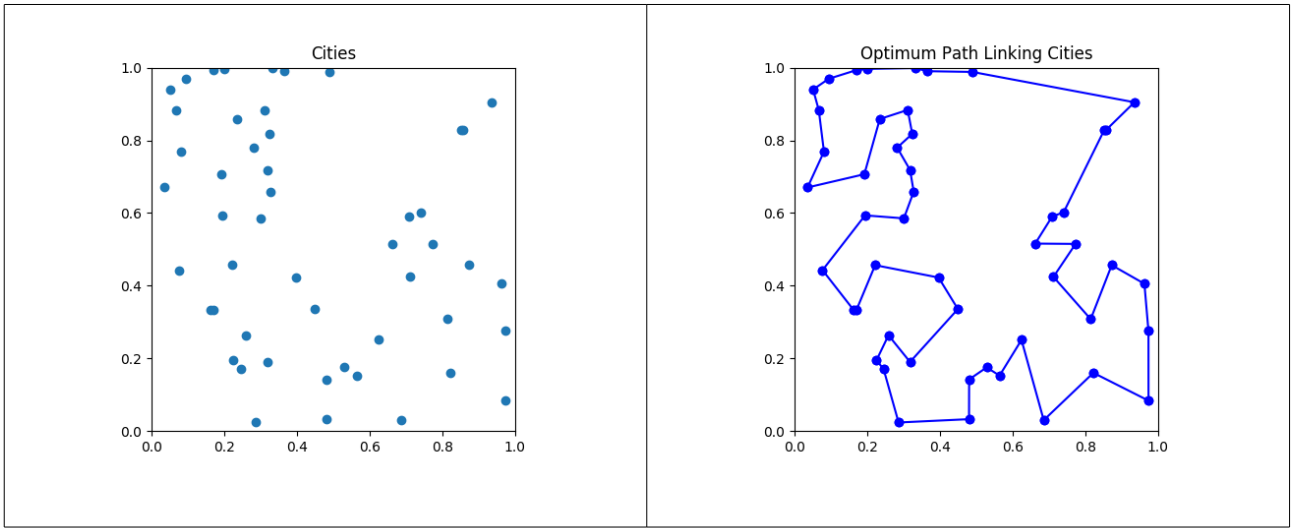



*Illustration 10: Graphical Solution to Traveling Salesman Problem for 50 Cities*

The process of annealing starts with a path which simply lists all of the cities in the order their positions were randomly selected. On each temperature step, the simulated annealing heuristic considers some neighbouring state, s`, of the current state, and probabilistically decides between moving the system to state, s`, or staying in state s. These probabilities ultimately lead the system to move to states of lower energy. Typically this step is repeated until the system reaches a state that is good enough for the application, or until a given computation budget has been exhaused.

The probability of making the transition from the current state s to a candidate state s` is specified by an acceptance probability function, P, that depends on the energies of the two states, e and e`, and on a global time-varying parameter T called the temperature. States with a smaller energy are better than those with a greater energy.

# Applications

Optimisation:

- Mechanics
  - Rigid body dynamics
  - Aerospace engineering
- Economics & Finance
  - Game theory
  - Microeconomics: utility maximisation problem
  - Resource allocation
  - Portfolio planning
- Electrical Engineering
  - Optimisation of microwave components & antennas
- Civil Engineering
  - Cut and fill of roads
  - Resource leveling and schedule optimisation
- Operations Research
  - Many types of scheduling, e.g. crew rosters, school time-tabling
- Control Engineering
  - Modern controller design
- Geophysics
  - Geophysical parameter estimation problems
- Molecular modeling
  - Conformational analysis
- Computational systems biology
  - Optimal experimental design
  - Maximal possible yields of fermentation products
  - Parameter estimation in biochemical pathways

# References

## Papers / Books

Although not directly referenced in the paper, the following texts were used extensively:

**Graduate:**

Oppenheim, A. and Schafer, W. (2013). Discrete-Time Signal Processing

Kutz, N. (2013). Data-Driven Modeling & Scientific Computation: Complex Systems & Big Data

Bishop, M. (2006). Pattern Recognition and Machine Learning

Hastie, T., Tibshirani, R. and Friedman, J. (2008). The Elements of Statistical Learning

LeVeque, R. (2013). High Performance Scientific Computing

Barba, L. A. (2014). Practical Numerical Methods with Python

LeCun, Y., Bottou, L., Bengio, Y. and Haffner. (1998). Gradient-Based Learning Applied to Document Recognition

Turek, D. (2004) Design of Efficient Digital Interpolation Filters for Integer Upsampling

Jha, R.G. (2012). On the Numerical Simulations of Feynman's Path Integrals using Markov Chain Monte-Carlo with Metropolis-Hastings Algorithms

**Undergraduate:**

***Calculus:***

Fowler, J. and Snapp, B. (2014). MOOCulus

Bonfert-Taylor, P. (2015). Complex Analysis

Blanchard, P. and Devaney, R. (2011). Differential Equations

***Linear Algebra:***

Klein, P. (2013). Coding the Matrix: Linear Algebra through Applications to Computer Science

Strang, G. (2006). Linear Algebra and Its Applications, 4th Addition

***Probability & Statistics:***

Conway, A.(2012). Statistics One

Johns Hopkins Specialisation (2014). Data Science

***Physics:***

Krauth, W. (2006). Statistical Mechanics: Algorithms and Computation

Aspuru-Guzik, A. (2017). The Quantum World

***Economics & Financial Mathematics:***

Zivot, E. (2016). Computational Finance and Financial Econometrics

Cvitanic, J. and Zapatero, F. (2004). Intro to the Economics and Mathematics of Financial Markets

Rangel, A. (2016). Principles of Economics with Calculus

***Electronics:***

Prandoni, P. and Vetterli, M. (2008). Signal Processing for Communications

Agarwa, A. and Lang, J. (2005). Foundations and Digital Electronic Circuits

***Computation and Algorithms:***

Guttag, J. (2013). Introduction to Computation and Programming Using Python

Sedgewick, R. and Wayne, K. (2011). Algorithms

***Additional:***

Wikipedia (multiple links)

Linear Algebra (scipy.linalg) https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Signal Processing (scipy.signal) https://docs.scipy.org/doc/scipy/reference/signal.html

Machine Learning: https://scikit-learn.org/stable/auto_examples/decomposition/plot_pca_iris.html

Data Analysis: https://pandas.pydata.org/

Graphs / Networks: https://networkx.github.io/

Simulated Annealing: https://www.fourmilab.ch/documents/travelling/anneal/

## Codes

The following codes were used to generate the various plots in the paper.

01_TaylorSeries_SineWave.py

02_FourierSeries_SquareWave.py

03_Draw_Arbitrage.py

04_BellmanFord_Aribtrage.py

09a_ModernPortfolioTheory.py

09b_MPT_Functions.py

10_SimulatedAnnealing_TravelingSalesman.py