

Euler, “our jewel”, Numerical Methods & Optimisation

Gavin Conran

Introduction

This revision / background paper introduces the Swiss mathematician, physicist, astronomer and engineer, **Leonard Euler**, viewed by many as the finest mathematician who ever lived and, to make him familiar with computer scientists, the creator of the mathematical notation to describe a function, $f(x)$. The paper starts with a discussion on what Richard Feynman referred 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. There is a coded example that uses Taylor’s Method and the Crank-Nicklson implicit finite difference scheme to simulate the diffusion of heat along a metal rod, as introduced in the Fourier paper.

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 switch to Euler’s key role in graph theory and then hand over to more modern day Graph algorithms for **discrete** optimisation. For constrained optimisation there is a discussion on the relationship between Graph algorithms and Linear Programming. Lagrange Multipliers, a mathematical technique used in optimisation, especially in Economics, and Jacobian and Hessian matrices, tools used generally when combining linear algebra and calculus, are also explored in the context of constrained optimisation.

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-hard problem of the Travelling Salesman Problem is solved using simulated annealing.

Euler's Formula

(Feynman referred 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

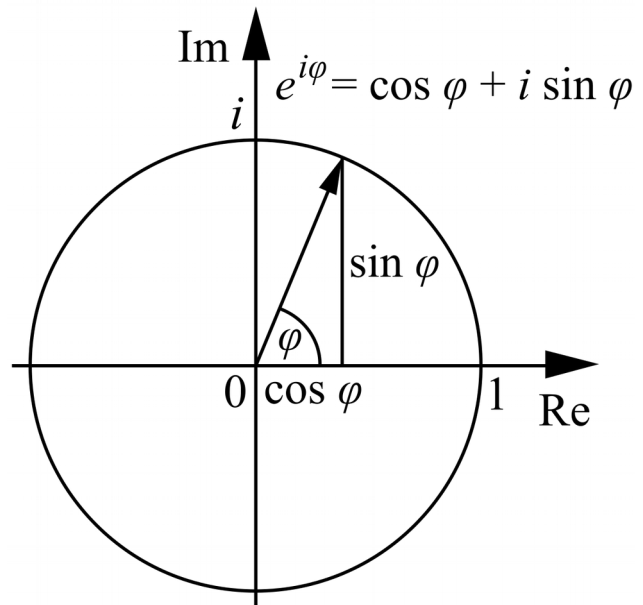
x is given in Radians

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

$$e^{i\pi} + 1 = 0$$

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 φ 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

$$x = \Re z$$

$$y = \Im z$$

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

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

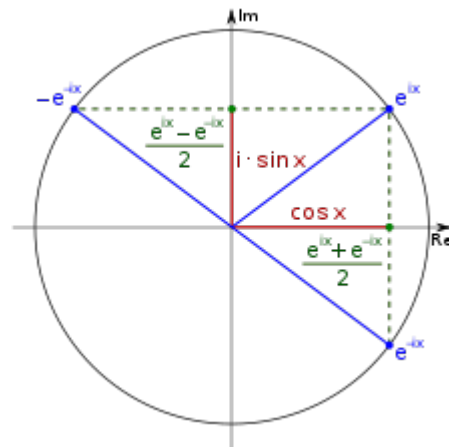
It is worth noting that the Polar Form simplifies the mathematics when used in multiplication or powers of complex numbers

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

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

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



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 exists.
- 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** :

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

This is commonly written as:

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

where τ is some value between t and $t + \Delta t$

which can also be written

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

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

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

Example: Binomial Series

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

Example: Trig Functions

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

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.

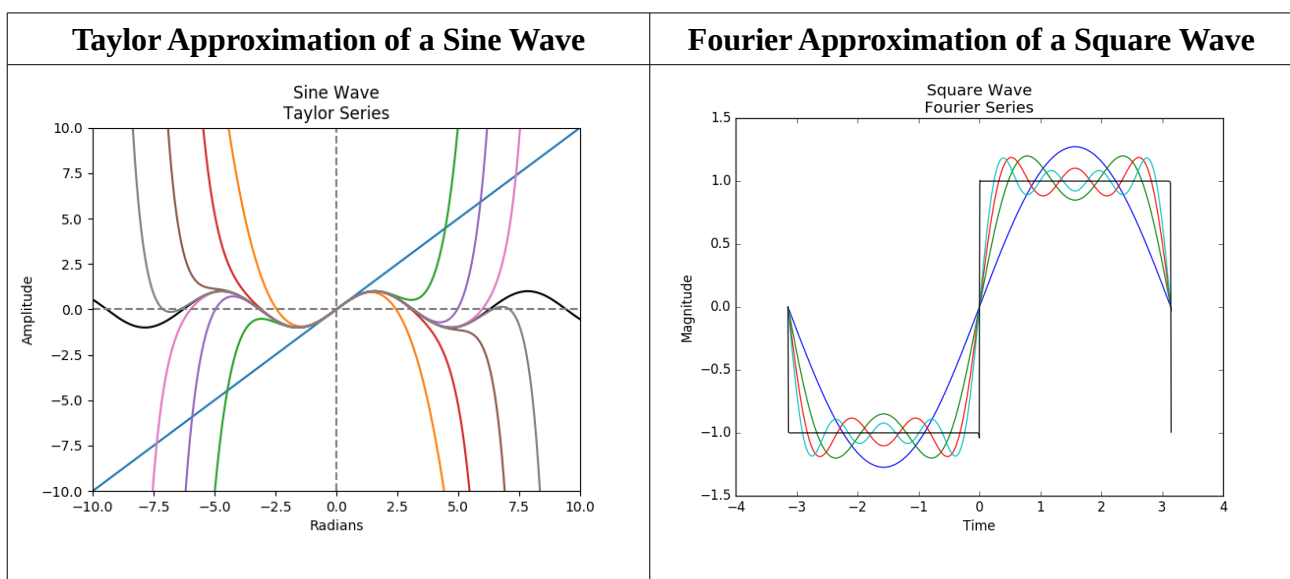


Illustration 1: Taylor Approximation vs. Fourier Approximation

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:

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

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:

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

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

$$y_{i+1} = y_i + a k_1 + b k_2$$

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 stability, 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:

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

where $y(t_{k+1})$ is the exact solution and

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

The Global (cumulative) Error is given by

$$E_k = y(t_k) - y_k$$

where $y(t_k)$ is the exact solution and

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, ϵ_k	Global Error, E_k
Euler	$O(\Delta t^2)$	$O(\Delta t)$
2 nd Order Runge-Kutta	$O(\Delta t^3)$	$O(\Delta t^2)$
4 th 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 Δt . Thus, it is tempting to conclude that higher accuracy is easily achieved by taking smaller time steps Δt . 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.

Numerical Solution using the Taylor Series

It is possible to find a numerical solution to the Heat Equation using the Crank-Nicolson method, a second order, implicit scheme derived from the Taylor Series approximation of the Heat Equation. The Crank-Nicolson scheme is used by professional Applied Mathematicians to solve Parabolic PDEs in various fields, such as financial mathematics. The RHS image of Illustration 2 shows a solution to the heat equation obtained by using the Taylor Series but, instead of both boundary conditions being Dirichlet as is with the Fourier Solution, this solution has a Dirichlet (LHS) and a Neumann (RHS) boundary condition.

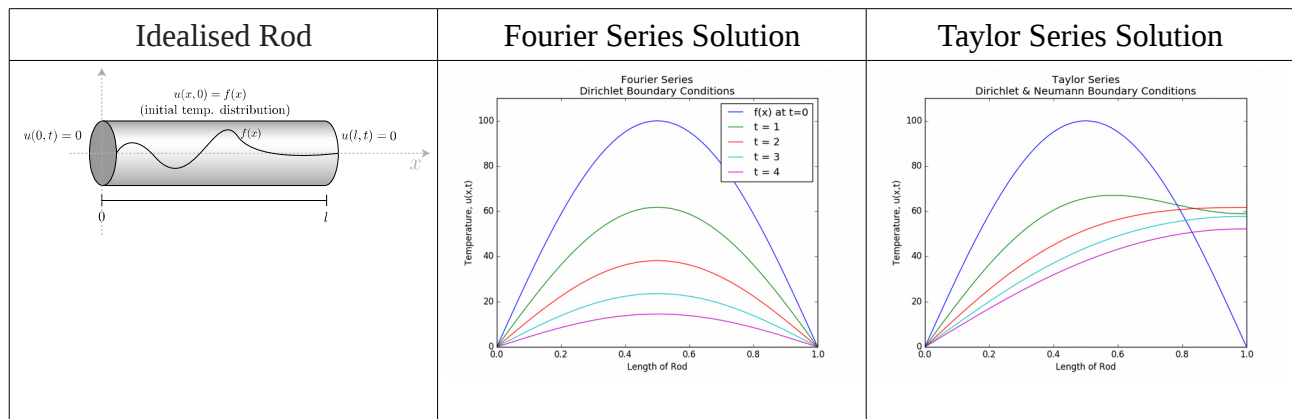


Illustration 2: Graphical Solutions to the Heat Equation

Optimisation

- Continuous:
 - Unconstrained: Machine Learning
 - supervised: Gradient Descent
 - unsupervised: Expectation-Maximisation
 - Constrained: Lagrange multipliers
- Discrete:
 - Constrained:
 - Graphs
 - Linear Programming
 - Unconstrained:
 - Simulated Annealing

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

In the simplest setting, the learning problem consists in finding the value W that *minimises* $E_{train}(W)$. 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 E_{test} and the error rate on the training set 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 reasonably 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

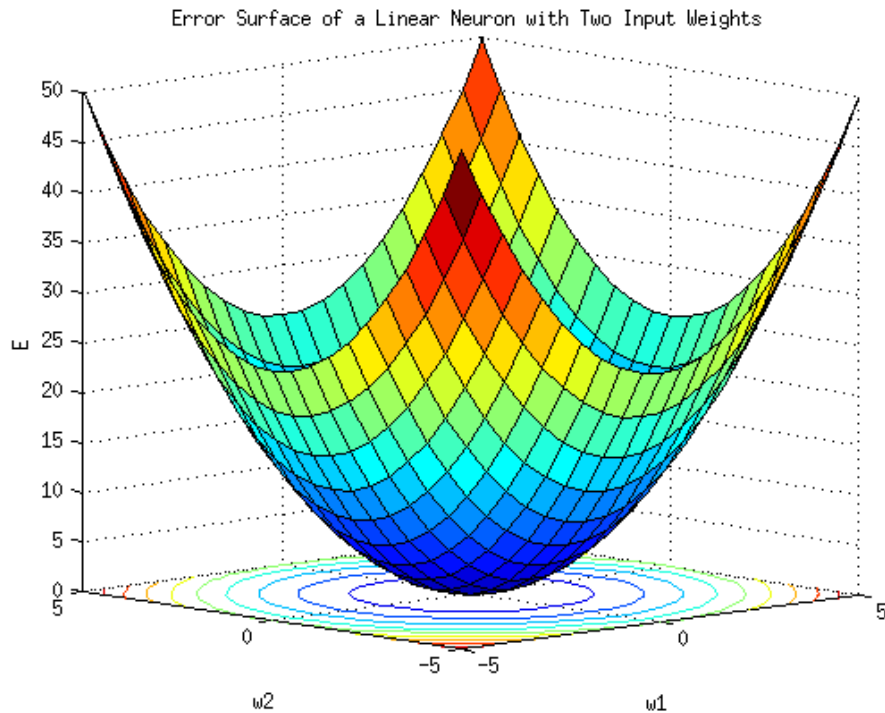


Illustration 3: Error Surface of a Linear Neuron with 2 Input Weights, w1 & w2

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 recognition).

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 X_n is a vector representing the output of the module

W_n is the vector of tunable parameters in the module (a subset of W)

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

$\frac{\partial F(W_n, X_{n-1})}{\partial X}$ 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 outputs 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 example in Illustration 4 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 **Maximum Likelihood solutions** for models having **latent variables**. We denote the set of all observed data by \mathbf{X} and similarly we denote the set of all latent variables by \mathbf{Z} . The set of all model parameters is denoted by θ , and so the **log likelihood function** is given by

$$\ln p(\mathbf{X}|\theta) = \ln \left\{ \sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\theta) \right\} .$$

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

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 θ^{old} , then a pair of successive E and M steps gives rise to a revised estimate θ^{new} . The algorithm is initialised by choosing some starting value for the parameters θ_0 .

E-Step:

We use the current parameter θ^{old} to find the posterior distribution of the latent variables given by $p(\mathbf{Z}|\mathbf{X}, \theta^{old})$. 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 $Q(\theta, \theta^{old})$ is given by

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

M-step:

We determine the revised parameter estimate θ^{new} by maximising this function

$$\theta^{new} = \arg \max_{\theta} Q(\theta, \theta^{old}) .$$

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

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

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

where

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

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:

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

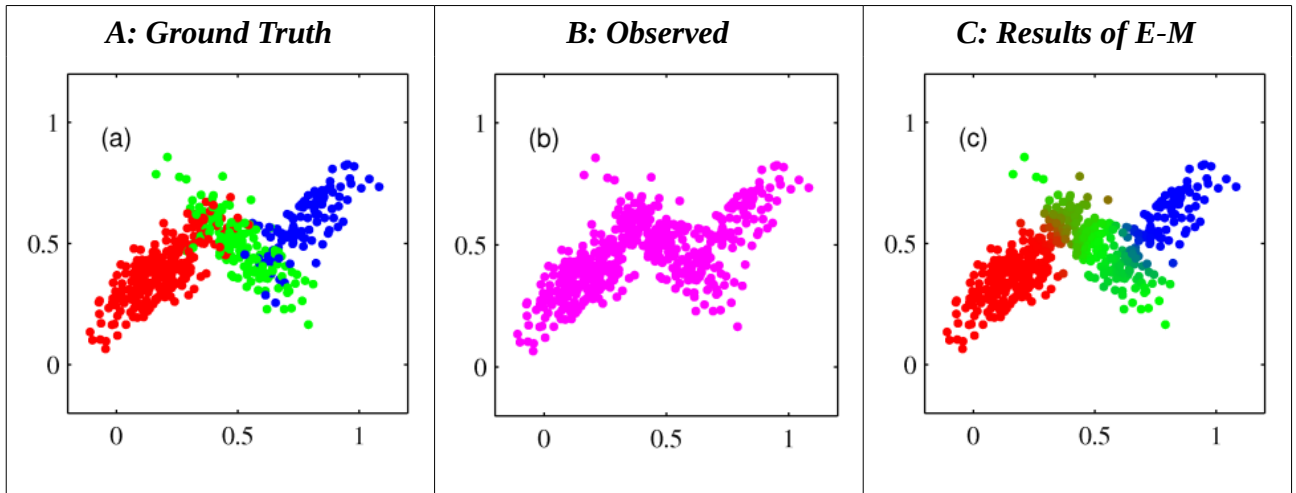


Illustration 4: EM algorithm performed on mixture of Gaussian data, \mathbf{X} .

In Illustration 4, **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 \mathbf{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.

Intuition

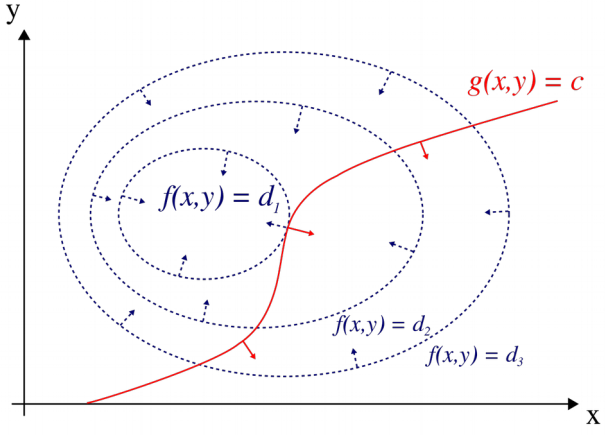
	<ul style="list-style-type: none"> • At a maximum, $f(x,y)$ cannot be increasing in the direction of any neighbouring point where $g=0$. • If it were, we could walk along $g=0$ to get higher, meaning that the starting point wasn't actually the maximum. • We want points (x,y) where $g(x,y)=0$ and $\nabla_{x,y} f = \lambda \nabla_{x,y} g$ 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 auxiliary 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 2: 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:

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

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

Example:

$$\begin{array}{l} \text{Maximise } f(x, y) = x + y \\ \text{Subject to } x^2 + y^2 = 1 \end{array}$$

Step 1: The Lagrange Function is:

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

after substituting the equations, we get:

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

Step 2: calculate the Gradients

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

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:

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

Step 3: The first two equations yield:

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

By substituting 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:

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

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

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

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

Jacobian & Hessian Matrices

Jacobian

(Matrix of 1st Order Partial Derivatives of a **Vector**-valued function)

Suppose $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a function which takes as input the vector $x \in \mathbb{R}^n$ and produces, as output, the vector $f(x) \in \mathbb{R}^m$ then the Jacobian Matrix, J , of f is an $m * n$ matrix:

$$J = \left[\frac{\partial f}{\partial x_1} \dots \frac{\partial f}{\partial x_n} \right] = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

or, component wise

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

- If the function is differentiable at point X
- Then the Jacobian Matrix defines a linear map, $\mathbb{R}^n \rightarrow \mathbb{R}^m$, which is the best point wise linear approximation of the function, f , near the point X . This linear map is thus the generalisation of the usual notion of a derivative and is called the **Derivative** or the **Differential of f at X** .
- if $m = n$ the Jacobian Matrix is a square matrix and its Determinant is the **Jacobian Determinant of f** . It carries important information about the local behaviour of f .
- f in the neighbourhood of point X has an Inverse Function that is differentiable iff the **Jacobian Determinant** is non-zero at X .

Example: Consider the function, $f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$, given by:

$$f(x, y) = \begin{pmatrix} x^2 y \\ 5x + \sin(y) \end{pmatrix}$$

Then we have

$$f_1(x, y) = x^2 y$$

and

$$f_2(x, y) = 5x + \sin(y)$$

And the Jacobian Matrix, J , of f is

$$J_f(x, y) = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{pmatrix} = \begin{pmatrix} 2xy & x^2 \\ 5 & \cos(y) \end{pmatrix}$$

And the Jacobian Determinant is

$$\det(J_f(x, y)) = 2xy \cos(y) - 5x^2$$

Hessian

(Square Matrix of 2nd Order Partial Derivatives of a **Scalar**-valued function)

Describes the Local Curvature (remember it is a 2nd order derivative) of a function of many variables.

Suppose $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a function taking as input a vector, $x \in \mathbb{R}^n$, and outputting a scalar, $f(x) \in \mathbb{R}$; if all 2nd Partial Derivatives of f exist and are continuous over the domain of the function, then the Hessian Matrix, H , of f , H_f , is a square $n * n$ matrix, usually defined and arranged as follows:

$$H_f(X) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \dots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \dots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

or component wise

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

The Hessian Matrix is related to the Jacobian Matrix by:

$$H_f(X) = J(\nabla f(X))^T$$

Note: H_f is a matrix with functions as entries and it is meant to be evaluated at some point, X , this is sometimes called a Matrix-valued Function.

Applications

Application 1: 2nd Derivative Test

IF Hessian is +ive definite (+ive for every non-zero column vector) at x

THEN f attains an isolated Minimum at x .

IF Hessian is -ive definite (-ive for every non-zero column vector) at x

THEN f attains an isolated Maximum at x .

IF Hessian has both +ive and -ive Eigenvalues

THEN x is a Saddle Point for f .

Application 2: Quadratic Approximations of Multi-Variate Functions

Hessian Matrices are used in large scale Optimisation problems within Newton type methods because they are the Coefficient of the Quadratic Term of a local Taylor Expansion of a function, that is:

$$y=f(x+\Delta x)\approx f(x)+\nabla f(x)^T\Delta x+\frac{1}{2!}\Delta x^TH(x)\Delta x$$

where ∇f is the gradient $(\frac{\partial f}{\partial x_1}\dots\frac{\partial f}{\partial x_N})$

Application 3: Image Processing

The Hessian Matrix is commonly used for expressing Image Processing Operators in field of Computer Vision.

Bordered Hessian

Used for the 2nd derivative test in certain Constrained Optimisation problems. Given a function, f , but adding a constraint function, g , such as $g(x) = c$, the bordered Hessian of the Lagrange Function, $L(x, \lambda) = f(x) + \lambda[g(x) - c]$ is:

$$H(L) = \begin{pmatrix} \frac{\partial^2 \lambda}{\partial \lambda^2} & \frac{\partial^2 \lambda}{\partial \lambda \partial x} \\ (\frac{\partial^2 \lambda}{\partial \lambda \partial x})^T & \frac{\partial^2 \lambda}{\partial x^2} \end{pmatrix} = \begin{pmatrix} 0 & \frac{\partial g}{\partial X} \\ (\frac{\partial g}{\partial X})^T & \frac{\partial^2 L}{\partial X^2} \end{pmatrix}$$

$$= \begin{pmatrix} 0 & \frac{\partial g}{\partial x_1} & \frac{\partial g}{\partial x_2} & \dots & \frac{\partial g}{\partial x_n} \\ \frac{\partial g}{\partial x_1} & \frac{\partial^2 L}{\partial x_1^2} & \frac{\partial^2 L}{\partial x_1 \partial x_2} & \dots & \frac{\partial^2 L}{\partial x_1 \partial x_n} \\ \frac{\partial g}{\partial x_2} & \frac{\partial^2 L}{\partial x_2 \partial x_1} & \frac{\partial^2 L}{\partial x_2^2} & \dots & \frac{\partial^2 L}{\partial x_2 \partial x_n} \\ \vdots & \vdots & & \ddots & \vdots \\ \frac{\partial g}{\partial x_n} & \frac{\partial^2 L}{\partial x_n \partial x_1} & \frac{\partial^2 L}{\partial x_n \partial x_2} & \dots & \frac{\partial^2 L}{\partial x_n^2} \end{pmatrix}$$

- If there are m Constraints Then the Zero in the upper-left corner is an $m * m$ block of zeros
- 2nd derivative test consists here of sign restrictions of the determinants of a certain set of $n-m$ submatrices of the bordered Hessian
 - Normal 2nd order derivative test rules cannot apply here since a bordered Hessian can neither be -ive / +ive definite

Graph Algorithms and Linear Programming

Note: It is also possible to solve Constrained Optimisation problems using Graph Algorithms, such as, Max Flow, or Linear Programming which may utilise the Simplex Algorithm.

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-hard Problem

The travelling salesman problem is a problem in graph theory requiring the most efficient, i.e. least total distance, Hamiltonian cycle a salesman can take through each of N cities. No general method of the solution is known, and the problem is NP-hard. That said, it can be solved using Simulated Annealing, as shown in Illustration 5 below.

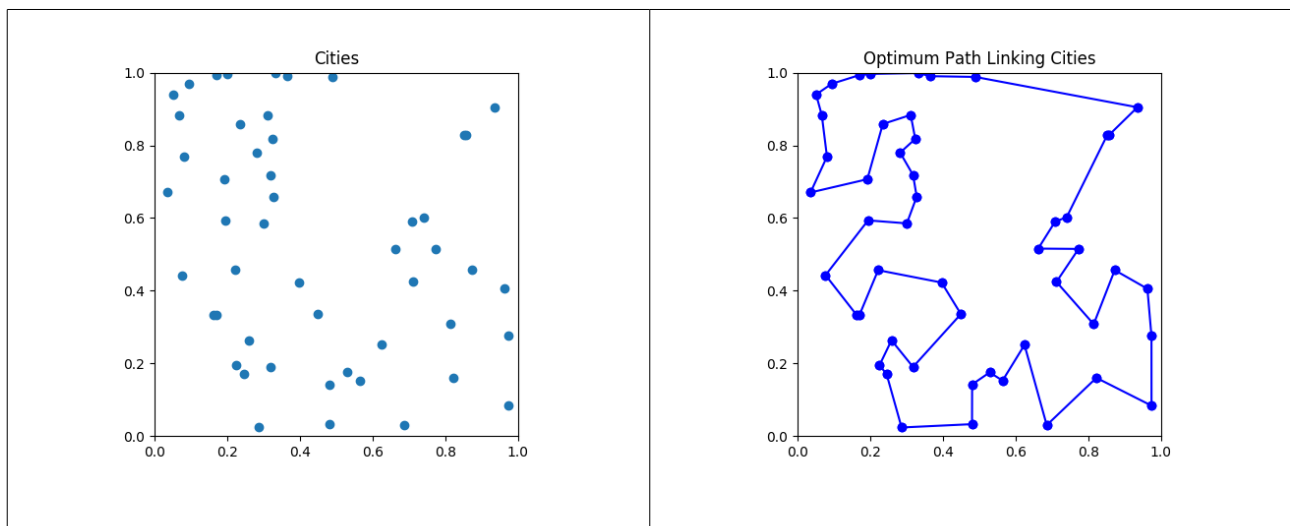


Illustration 5: Graphical Solution to Traveling Salesman Problem for 50 Cities

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 exhausted.

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

Lagrange Multipliers:

- Managerial Economics

References

Papers / Books

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

Descartes, R. (1644). Principles of Philosophy

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

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

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

Gottlieb, S. (2004). Euler's Method, Taylor Series Method, Runge-Kutta Methods, Multi-Step Methods and Stability

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

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

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

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

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

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

Wikipedia (multiple links)

<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_FourierSeries_HeatEquation

04_TaylorSeries_HeatEquation

10_SimulatedAnnealing_TravelingSalesman.py