Disclaimer: This essay entry is written by an enthusiast of the FFT in the interest of documenting, collating and sharing the learning on the subject providing some use-cases as reference. It is meant as an informative piece that can help people of similar interests understand the subject better with a variety of interesting branches compiled into a single hopefully-fun-to-read document. The author is neither a mathematician nor historian, and shall not be held liable for factual inaccuracies nor losses faced in application of the presented algorithms, even though periodic reviews and other efforts will be made to ensure at least a certain level of accuracy and quality being maintained.

Funding: This paper was completely self-funded by the author - Gene Boo.

Conflict of Interest: The author declares that there is no conflict of interest.

Received April, 2020; | Revised April, 2020; | Accepted April, 2020

DOI: xxx/xxxx

A GENERAL INFORMATIVE ESSAY FOR GENERAL ENTHUSIASTS AND PRACTITIONERS ON FFT IN STATISTICS & FINANCE

Fast Fourier Transform as an Engine for the Convolution of PDFs and Black-Scholes Option Pricing Using Python

Gene Boo*[1]

Correspondence
*Gene Boo Email: gene.boo@aol.com

Abstract

This paper is an attempt by an FFT enthusiast to document *one of the most magical mathematical engines from the end of the previous millennium*, which created the possibility of sound/video/photo editing in the frequency domain, with a focus on statistical convolutions and an example on constant volatility Black-Scholes pricing with minimal pre-transformations.

KEYWORDS:
Fast Fourier Transform; FFT; FT; DFT; Signal; Convolution; Cyclical; Probability Density Function; Option Pricing; Quantitative Finance; Black Scholes; Garman Kohlhagan; History; Frequency; Time; Music; Digital Signal Processing; DSP; Python; NumPy; SciPy; MatPlotLib; VBA; Excel

1 | INTRODUCTION

The author of this paper is an employee in the Malaysian banking sector, who started his quantitative training and career in Brussels, Belgium, where he had been exposed to the wondrous FFT algorithm in the course of pursuing his hobbies - electronic music production and video game algorithms. As of this moment of essay-writing, the frightful Covid-19 pandemic is still ongoing, and us in Malaysia, like the rest of the world, are in a locked-down state - what better moment to use our time saved from commuting to and from work, to pen a little bit about the FFT in hopes of seeing it get picked up for more common use in the Asian financial sector.

The banking and actuarial sectors often rely on probabilistic models for the purposes of pricing and risk management, and the most commonly encountered distributions in financial mathematics in Asia are probably the Normal and Lognormal distributions.

With Normal distributions, convolution is rather easy as the sum of N Normals is really just a single Normal with the means and variances of the variates summed up. The same cannot be mentioned for the Lognormal. The Lognormal distribution is frequently used in Geometric Brownian Motion models for option pricing - such as Black-Scholes. If $ln(X)$ follows a Normal distribution, X follows a Lognormal distribution.

It is well known that where there are many summed Lognormals, the final distribution tends toward the Normal, although not quite. With lower sums, the distribution looks "Lognormal-ish", but it is not quite there neither. What if the model requires an arbitrary sum of Lognormals - e.g. Basket Options? Much research has been conducted on this, where some of the more recent ones attempt approximations like Gauss-Hermite expansion, and application of Laplace Inversions using contour integration[1] - complicated algebra!

[1]https://arxiv.org/pdf/1606.07300.pdf

The takeaway from the previous paragraph is that as simple as Lognormals seem to be, summing them is not trivial. While it is great as a research topic, it only serves as a bottleneck in industries, which then turns to brute force approaches such as Monte Carlo, yielding oscillating results - definitely a problem in post-utility such as computing capital allocation weights, and besides, the engine is most definitely not one for the impatient.

If we had a reliable engine that can sum up not just Lognormals but any arbitrary shape describing random variates stably and very quickly - it could be an industry game-changer. The FFT algorithm when used as a convolution engine, does precisely that, and is quick and accurate as well. The FFT engine is capable of convolving any arbitrary signal, hence empirical distributions of any sort and other types of shape-functions can also be combined with precision. While frequently seen in science and engineering, its use is not yet popularised in the Asian financial sectors to date. However, with the description above, one can already imagine its potential use in various financial topics encountered today where probability distributions are frequented - option pricing, XVA estimation, credit loss probability estimation, operational risk compound loss distribution estimation, liquidity risk profiling etc. to name a few.

The author hopes that this introductory essay may serve to stir up some interest in the topic whilst remaining a fun-read, such that the banking industry may consider to start adopting FFT approaches in some of their modelling efforts - be it for combining pdfs, or obtaining more precise option prices other than those generated by Monte Carlo methods, to calculating huge covariance matrices, or obtaining stable tail-risk estimates such as VaR and Expected Shortfall for use in capital allocation etc.

It draws examples and concepts from the author's other hobbies (audio synthesis and Digital Signal Processing (DSP)) with some simple examples and Python 3 code. On the history section, the reader is cautioned not to put 100% confidence in its accuracy as there are varying versions that the author has encountered.

Real-world FFT topics and applications can involve very advanced mathematics and coding structures, so please do not let the simplicity of this essay cause an underestimation of its complexity and far-reached use.

On that note - let's dilly-dally no longer. Join us on our FFT journey - we hope you enjoy the ride!

2 | SOME INTERESTING HISTORY

The Fast Fourier Transform (FFT) is a well known and highly popular method in the world of engineering and physics and many other schools where signals are measured over time. We celebrate its discovery in 1965 by James Cooley and John Tukey, but actually, the theory of Fourier Transform (FT) and the concept of the FFT was penned long ago by the legendary Johann Carl Friedrich Gauss - of whom the Normal distribution was named after, displaying a case of Stigler's Law, as is the Fourier Transform to Joseph Fourier. Just to provide an inkling of its importance - without the FT and FFT, our world would not be so technologically driven as it is today - with computers, screens, rocketships, supercars, club music and remixes, computer animations, video games, fridges that can sense depletion and order your milk via the Internet, facial recognition software, fraud detection, encryption and compression - you name it. Hence that is why those born during the era of the Baby-boomers and Gen-X could observe the sudden technological jump, revolutionising the world - rendering a complete face-lift. This is all thanks to the skeleton key of FFT opening the portals of possibilities. This magical key opened the door to enable us to work on arbitrary signals and waveforms, allowing solutions to many previously-thought-unsolvable problems in almost all scientific and mathematical disciplines.

2.1 | The Wizards of FFT

Johann Carl Friedrich Gauss is purportedly the true inventor of the underlying recursive FFT algorithm, although there was a lack of hardware and software infrastructure in the very early 19th Century for its use to become popularised (Understatement-of-the-Millennium!) His 'FFT' work was then not as recognised as his other works, as it was way before its time, but it has since found its throne in the 20th and 21st Century (and probably way further to come). Gauss had initially used it as an interpolation tool for asteroid trajectories. This work was published posthumously.

IBM's James Cooley and Princeton's John Tukey published a paper in 1965, reinventing the FFT algorithm for use on a computer. Like much technology invented during the mid 20th Century, it had been primarily intended for military use, although Cooley was misdirected and informed that it was for crystallography. When their paper was published, it had been around the era of the Analog-Digital Converter (ADC), which conveniently implemented and spread the use of the algorithm. ADCs have today shrunk to the size of mini metal-oxide-semiconductor microchips, and are commonly found soldered to or directly integrated within music equipment such as modern analogue synthesizers, which enable harddisk recording or waveform manipulation such as digital filtering (post synthesis), or simply to enable the sending of the signal pre-converted to digital form, to a computer or digital synthesizer to avoid cable losses and further processing on the receiver machine. Analogue signals weaken, distort or deteriorate if they have to traverse copper or gold wires - also hence fibre optic cables of Sony/Philips Digital Interface (S/PDIF) as the new platinum standard - retaining the quality of original signals much closer to perfection. All sound cards have an ADC built on. ADCs are of course integrated within all our mobile telecommunicative devices today to convert waveforms picked up by the microphone or light by the camera, into digitally sampled signals. This is complementary to Digital-to-Analogue Converters (DAC), whose purpose is to reverse the function - i.e. playback a digital waveform to speakers or generating the appropriate LED matrix for viewing signals on-screen.

However, as it is with most works of great transcendental mathematics, the idea behind the development of FFT is derivative of other ground works. We must thank a few more legendary 17th and 18th Century European 'Magicians' - Roger Cotes known for his Newton-Cotes Quadrature[2] formulae (enabling us to perform numerical integration at good precision using Lagrange basis polynomials), Leonhard Euler from whom constant e got its honoured name-sake "Euler's number" - yet another case of Stigler's Law, and Jean-Baptiste Joseph Fourier who further developed the trigonometric series works by Euler, Bernoulli (who is the actual discoverer of the constant e) and d'Alembert, into the Fourier Series (FS).

2.2 | Bernoulli's Contribution

Bernoulli discovered the constant e while experimenting with concepts on compound interest. He was also amongst the founders of complex logarithms.

2.3 | Cotes' Contribution

British mathematician Roger Cotes had a very important finding - $ix = ln(cos(x) + isin(x))$ - do compare it to the Euler formula below. We all know how $e^{ln(x)} = x$, thus it gets interesting.

2.4 | Euler's Contribution

Euler predated Gauss, and he is well known for his equations $e^{i\pi} + 1 = 0$ and $e^{i\pi} = cos(\pi) + isin(\pi)$. These are the Euler identity and Euler formula respectively - usually π is represented generally by x for the Euler formula, but by substituting π for x as a special case, the identity equation would be the outcome. Just for the reader to draw a parallel, the Fourier Transform is often formulated as $\mathcal{F}\{f(t)\} = F(x) = \int_{-\infty}^{\infty} f(t)e^{-2\pi itx}dt$ where $i = \sqrt{-1}$. One can immediately see the resemblance of the 2, both involving constants i, e and π, hence a notion about the logical development of the Fourier formula.

The journey from the Euler formula of a circle traversing through the real and complex plain, to derivation of the identity simultaneously kick-started branches of many other discoveries of mathematics as it establishes links between the constants e, π, 1, i, and functions of $\sqrt{}$ and exponentiation, primarily in the further development of calculus in higher dimensions, without which we would not be able to solve many physics problems, nor develop technology as we see today such as the emergence of computers for use with designs and analyses. The trigonometric functions relate the triangle to the circle, and knowledge of various basic identities[3] can help with algebraic manipulations.

[2] https://mathworld.wolfram.com/Newton-CotesFormulas.html
[3] http://math2.org/math/trig/identities.htm

2.5 | Fourier's Contribution

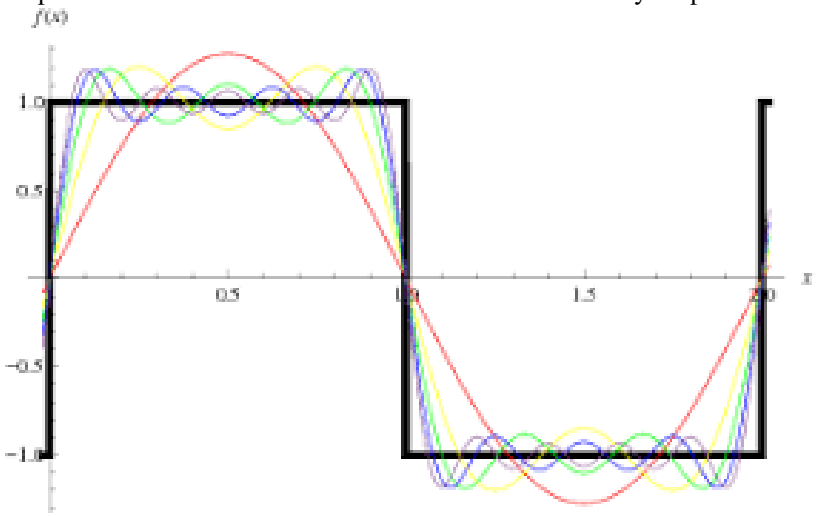
Fourier had discovered in the late 18th to early 19th Century that he could represent any continuous shape using the combination of trigonometric functions $\cos(x)$ and $\sin(x)$ in the form of a trigonometric series, having built upon the initial works of Euler, Bernoulli and d'Alembert. The purpose was to solve the heat equation $\frac{\partial u}{\partial t} = \alpha(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2})$, which apart from measurement of temperature distribution also found other general applications in today's stochastic mathematics involving Brownian Motion via the Forward Kolmogorov[4] (or Fokker-Planck[5]) Equations - which in turn provide an engine to solve various partial differential equations (pde), a recent famous one being the Black-Scholes equation $\frac{\partial V}{\partial t} + \frac{\sigma^2 S^2}{2}\frac{\partial^2 V}{\partial S^2} = rV - rS\frac{\partial V}{\partial S}$ for option pricing.

3 | THE FOURIER SERIES

The FS, making up a study known as harmonic analysis, expands a periodic function $f(x)$ in terms of the infinite sum of $\cos(x)$ and $\sin(x)$, by exploiting the orthogonality of the 2 functions. One could form any 2D shape inclusive of waveforms such as the sawtooth, square[6] and triangular - widely used in electronic music synthesizers. There are very creative videos on Youtube [7] that trace the silhouette of Matt Groening's cartoon character Homer Simpson using a combination of Fourier analysis and Ptolemy's system of epicycles - i.e. circle mapped onto the circumference of a larger 'mother' circle recursively (looks like the cross section of an engine turning a wheel). Interestingly the idea of using epicycles was already present during B.C. times, used in early astronomic studies by Hipparchian, Ptolemaic, and Copernican systems. One can visualise for example, the trajectory of the Earth circling the Sun, with another circle made by the moon circling the Earth.

3.1 | Gibb's Phenomenon

The approximation by FS of non-continuous functions such as the square waveform tend to yield 'ringing' near the edges of the discontinuity. This is known as the Gibb's Phenomenon. Graphically, we see the sinusoids near the discontinuity take a larger amplitude overshoot in order to U-turn for the discontinuity drop or rise. We feature here a picture of it cited from Wolfram[8].



3.2 | The Journey from FS to FT

$$f(x) = 0.5a_0 + \sum_{n=1}^{\infty} a_n cos(nx) + b_n sin(nx)$$

[4]https://www.sciencedirect.com/topics/mathematics/kolmogorov-equation
[5]https://demonstrations.wolfram.com/BrownianMotionIn2DAndTheFokkerPlanckEquation/
[6]https://demonstrations.wolfram.com/FourierCoefficientsOfASquarePulse/
[7]https://www.youtube.com/watch?v=D-ogjQP_TBs
[8]https://mathworld.wolfram.com/GibbsPhenomenon.html

where

$$a_0 = \frac{\int_{-\pi}^{\pi} f(x)dx}{\pi}$$
$$a_n = \frac{\int_{-\pi}^{\pi} f(x)cos(nx)dx}{\pi}$$
$$b_n = \frac{\int_{-\pi}^{\pi} f(x)sin(nx)dx}{\pi}$$

$\{a_0, a_n, b_n\}$ form the series coefficients, $sin(x)$ and $cos(x)$ form a complete orthogonal system over $[-\pi, \pi]$ and $n = 1, 2, 3, ...$ [9]. Note that the formulation is all done for real numbers. Orthogonality (perpendicularity) is important as it means the decomposition is made up of independent functions, hence combination can be made directly - same principle as Eigenvalue decomposition. Hence the next step is to make it complex. Recalling Euler's formulae - which provide the way to formulate an exponential as weighted trigonometric functions cos(x) and sin(x) :

$$e^{ix} = cos(x) + isin(x)$$
$$e^{-ix} = cos(-x) + isin(-x)$$

Lead to:

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

Hence when applied to the FS:

$$f(x) = 0.5a_0 + \sum_{n=1}^{\infty} a_n cos(nx) + b_n sin(nx)$$
$$= 0.5a_0 + \sum_{n=1}^{\infty} \frac{a_n - ib_n}{2} e^{inx} + \frac{a_n + ib_n}{2} e^{-inx}$$
$$= \sum_{-\infty}^{\infty} c_n e^{inx}$$

where complex coefficients $c_n = \frac{\int_{-\pi}^{\pi} f(x)e^{-inx}dx}{2\pi}$, $n = 0, \pm1, \pm2, ...$, making it algebraically simpler than the original equation.

Thanks to the effort of many FT-savvy netizens who are also coding wizards, we can now develop an intuition to these formulae by viewing animation of functions mapped to epicycles.[10]

The idea behind FS is to decompose signals into their component frequencies and harmonics (higher frequencies usually of lower amplitude/volume that are integer multipliers on the base frequency), that make up the full frequency, by essentially mapping the signal in time domain to epicycles and integrating it by simultaenously rotating all these circles at their given independent frequencies[11]. Applying the inverse FT (iFT) allows us to get back the signal. Hence for applications like sound editing/cryptography/blending/filtering, it is easy to just apply the FT, work on the frequency representation, then apply iFT to get the transformed signal in time domain. This also opens up a doorway to the world of convolution as will be featured later, as it is also a transformation to the frequency, by a function that happens to be the FT of the other signal.

While this version of the FS starts to resemble the FT, it is still limited to periodic functions and discrete frequency spectrum. To create a continuous version, we let $x = wt$ and change dx to dt. Now we have more control of the variable, by looking at it as a frequency w on a period t - which of course could have been done that way from the beginning. Now, to 'emulate' continuity, we let the period T tend to ∞. That way, w - the fundamental frequency will become an infinitesimally small dw, with n having to tend to ∞. $[n\ dw]$ tends to w, and c_n becomes $c(w)dw$ to preserve the integral as discrete tends toward continuous.

$$c(w)dw = \lim_{T\to\infty} \int_{\frac{-T}{2}}^{\frac{T}{2}} f(t)e^{iwt}dt$$

[9] https://mathworld.wolfram.com/FourierSeries.html
[10] https://www.youtube.com/watch?v=r6sGWTCMz2k
[11] https://en.wikipedia.org/wiki/Fourier_series#/media/File:Fourier_series_square_wave_circles_animation.gif

$$= \frac{dw}{2\pi} \int_{-\infty}^{\infty} f(t) e^{iwt} dt$$

This is the basic gist of getting from the FS to FT. Formally, there are a few more steps to perform and also some scaling with 2π to meet modern convention, but the idea on the development is clear.

4 | THE FOURIER TRANSFORM

Displayed here is the Hertz form definition of FT (forward FT, aka Analysis equation):

$$\mathcal{F}\{f(t)\} = F(w) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i t w} dt$$

where $i = \sqrt{-1}$ and iFT (aka Synthesis equation):

$$\mathcal{F}^{-1}\{F(w)\} = f(t) = \int_{-\infty}^{\infty} \mathcal{F}(w) e^{2\pi i t w} dw$$

4.1 | FT as a Convolution Engine

Convolution - one of the important usages of the FT:
Let $f(t)$ and $g(t)$ functions belong to the space of real numbers $\mathbb{R}^n$, with F and G being their respective FTs, and $h(z)$ be the linear convolution of f and g:

$$F(w) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i t w} dt$$

$$G(w) = \int_{-\infty}^{\infty} g(t) e^{-2\pi i t w} dt$$

The usual linear convolution formula:

$$h(z) = \int_{-\infty}^{\infty} f(t) g(z-t) dt$$

Applying the FT on $h(z)$ and Fubini's iterated integral theorem for double integrals:

$$H(w) = \mathcal{F}\{h(z)\} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (f(t) g(z-t) dt) e^{-2\pi i z w} dz$$

let $y = z - t$ and hence $dy = dz$ and reapplying Fubini:

$$= \int_{-\infty}^{\infty} f(t) \left(\int_{-\infty}^{\infty} g(z-t) e^{-2\pi i z w} dz \right) dt$$

$$= \int_{-\infty}^{\infty} f(t) \left(\int_{-\infty}^{\infty} g(y) e^{-2\pi i (y+t) w} dy \right) dt$$

$$= \int_{-\infty}^{\infty} f(t) e^{-2\pi i t w} \left(\int_{-\infty}^{\infty} g(y) e^{-2\pi i y w} dy \right) dt$$

$$= \left(\int_{-\infty}^{\infty} f(t) e^{-2\pi i t w} dt \right) \left(\int_{-\infty}^{\infty} g(y) e^{-2\pi i y w} dy \right)$$

$$H(w) = F(w)G(w)$$

showing that FT of the convolution function is simply an element-wise multiplication between the FTs of each function. Using FT to perform a convolution is known as cyclical or circular convolution, as it also has a wrap-around effect, being mapped to a circle. If we use the numerical FFT engine to convolve, we can see this easily if we take the exact N samples of the signal and perform a convolution directly. The output will not be as expected - a work-around known as 'zero-padding' which will be discussed later has to be in place.

Imagine the implications, for we can now specify convolutions by taking the iFT of the convolved $H(w)$ to get the new function that describes the convolution of $f(t)$ and $g(t)$. This means, a function that describes how each element of f affects each element of g, commutatively! In probability theory it would be equivalent to finding the combined density kernel of summing 2 random variates of arbitrary density kernels. In sound, it would be a mix-down of 2 signals i.e. like kick drum played together with a snare. Still, analytically performing the algebra and calculus may lead to unwieldy roadblocks for many types of functions, so the true power does not really lie here. Instead, the true power lies halfway in between - where if we have a numerical algorithm that can perform very quick FT and iFT, that takes away the analytical evaluation steps, allowing us to perform this exact same task to a very good approximation, without overly wrecking our brains.

4.2 | FFT Engine

FFT is that key. The algorithm computes the Discrete Fourier Transform (DFT) and inverse (iDFT) of a sequence - very quickly. It works by decomposing the sequence or signal into the frequency components that make it up. While DFT and its inverse can be applied directly mathematically, it quickly hits the 'curse of dimensionality' and hence rendering it not practical for larger scale problems.

If we were to approach DFT computation using linear algebra, we could create a square DFT matrix by specifying a transformation $W = \frac{w^{jk}}{\sqrt{N}}$ Vandermonde matrix (matrix containing geometric progression terms $V_{i,j} = \alpha_i^{j-1}$), of (N^2) dimensions holding powers of the complex exponential terms $w = e^{\frac{-2\pi i}{N}}$, and compute the matrix multiplication $X = Wx$, with x being a vector containing equi-distanced samples up to N points of the original signal and X being the Discrete-Time Fourier Transform (DTFT) vector also up to N points:

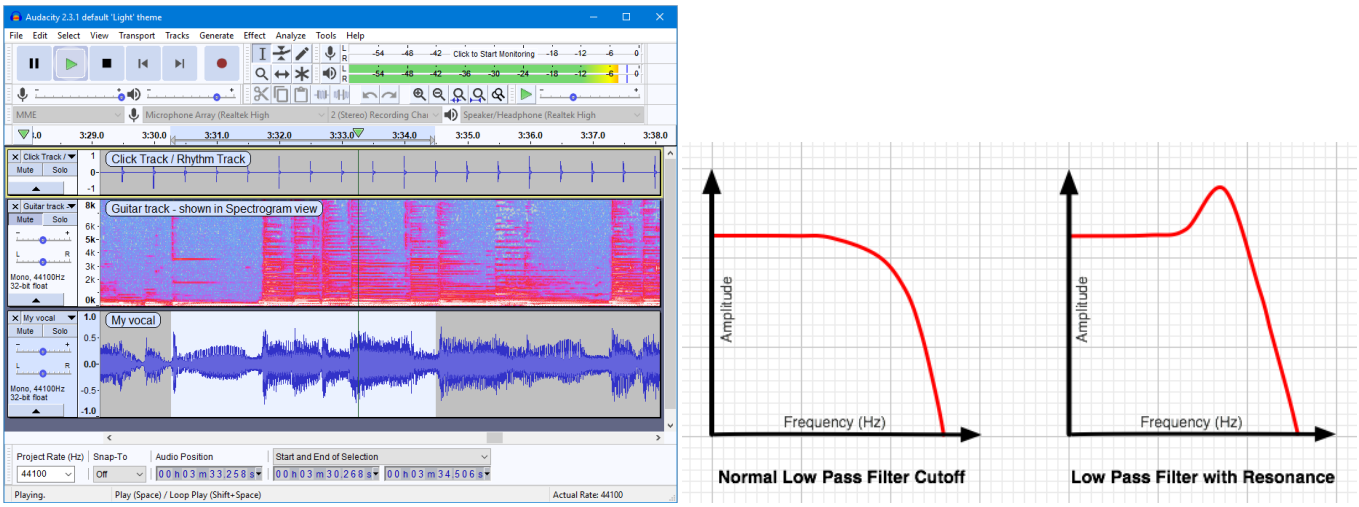
$$W = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{N-1} \\ 1 & w^2 & w^4 & \cdots & w^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & \cdots & w^{(N-1)^2} \end{pmatrix}$$

This is good for smaller N but gets computationally heavy as N grows large. In computer science and mathematics we say that the DFT computation takes on an order of complexity of $O(N^2)$, as the reader can observe in the formulation above - for each additional sample to be transformed, you have roughly its square more operations to perform. Note as well that we are dealing with complex matrix multiplications. Each row of W is really the decomposed component waveform as we see in an FS, and we are combining these with the original signal using matrix multiplication to get the summed products of each component waveform applied to all samples of the signal, which will make up the DTFT vector X of the signal. Since this is complex multiplication, we will have both real cos and imaginary sin numbers each element of X.

The top row in the matrix being $w^0 = 1$ multiplied and summed across the signal vector is really what is known as the 'total DC-offset or DC-bias' and the follow-up division by the scalar $\sqrt{N}$ will be the 'mean DC-offset', which measures the average value of the waveform over a single period. This is of course well related to the electrical concept of Direct Current and Voltage - like in Alternating Current (AC), we want the sum to be 0. Mathematically it is really equivalent to normalising an integral like dividing the Normal distribution by $\sqrt{2\pi}$, as it is the value of the total integral space. In sound engineering, we also want the DC offset to be 0, because a signal with DC-bias - representable by a waveform whose mean hovers above/below the 0 dB line, would be played back with potentially a lot of clicks and distortion, not a clean high-fidelity sound, unless of course the artist is creating a 'Vinyl-warmth Lofi Hip-hop' tune. Amplitude (volume) can also be lost, as the DC-offset affects the 'energy' of the system. Hence in remastering sessions using Sound Forge or Audacity, one might choose to remove the DC-offset of the waveform using normalisation and/or compression techniques - which really are applications of linear operations on the signal

vector. It is an understanding to keep in mind when designing filters like VST-plugins.

Eyeballing the Vandermonde matrix, one can observe the symmetry that exists across the diagonal (hint - just tilt your head slightly to the left and pretend the diagonal of W is a mirror). This of course means that there are operational reductions possible with a rephrasing of the solution. The typical FFT algorithm by Tukey and Cooley exploit observed attributes such as this, reducing the operations to a super efficient $O(Nlog(N))$.



Above on the left is a screenshot of free GPL Audacity[12] sound editing software. Observe the 2 different views - the one on the top is the 'spectrogram view' and the one below is 'waveform view'. Spectrogram is basically the plot of the FFT of a waveform signal, the various denser regions of horizontal lines are spread out in the harmonics that make up the waveform. The graphs on the right are 2 charts showing 2 parametric effects very often used in DJ-ing and electronic music production - namely the cut-off and resonance filters used on the Fourier-transformed signal in the frequency domain. Tweaking those parameters create some 'airplane whoosh' effect. The graphic is sourced from audio software Spectrasonics website[13].

4.3 | Radix-2 FFT by Cooley and Tukey

Cooley-Tukey FFT uses a recursive 'divide-and-conquer' strategy to break the problem of size N into smaller DFT problems of size N_i, where $N = \prod_{i=1}^{M} N_i$. Setting $M = 2$ for each decomposition iteratively is known as the Radix-2 case. This is coupled with $O(N)$ multiplications by complex roots of unity (de Moivre numbers - which are complex numbers that yield 1 when taking powers of n where n is a positive integer). To date, there are many FFT formulations that have emerged, taking on forms which have been designed for specific tasks, among some popular ones are Hexagonal FFT, QFT, Winograd FFT, etc. Many types of DFT algorithms involving only the real (or imaginary) plane exist as well like the Discrete Cosine Transform (DCT) - which finds good use in .JPG, .MPG, .MP3 file formats involving lossy compression. Lossy compression forgoes some frequencies present in the original data sample - e.g. generally inaudible harmonics, as opposed to lossless compression such as .FLAC audio format (which is approximately half the size of the original sample). Of course, reducing the rate of sampling (e.g. choosing 22 KHz of sampling as opposed to 44.1 or 48 KHz) gives rise to a compression as well, but at the expense of audible frequencies. There are Artificial Intelligence based routines that can 'bootstrap' based on the lower-frequency-sampled data and application of maximum likelihood methods and other filtration stacks, to construct artificially, the lost frequencies yielding a pseudo-original quality. This is however far from perfect, but getting popular and is a main marketing feature in some professional remastering software.

[12] https://manual.audacityteam.org/man/spectrogram_view.html
[13] https://support.spectrasonics.net/manual/Trilian/edit_page/filters/page21.html

The Radix-2 form of FFT is one of the more common forms of FFT. DFT:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2nk\pi i}{N}}$$

Radix-2 distributes recursively the computation into even- and odd-indexed samples.

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-\frac{4nk\pi i}{N}} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-\frac{(4n+2)k\pi i}{N}}$$

Furthermore, due to the periodicity of the complex exponential, $X_{k+\frac{N}{2}}$ can also be obtained from the above equation by substituting k for $k + \frac{N}{2}$ - whose output is a mirror of X_k with the odd-indexed reflected across the even-indexed.

$$X_{k+\frac{N}{2}} = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-\frac{4nk\pi i}{N}} - \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-\frac{(4n+2)k\pi i}{N}}$$

This makes the computation times a lot faster. Further improvements in speed is obtained by recycling intermediate results to simultaneously compute DFT output. The process can be simulated by hand using a Butterfly Data Diagram, where we split the DFT into 2 blocks of $\frac{N}{2}$ even- and odd-indexed points, and work out the DTFT of each signal point. Simultaneously, the sign is flipped and recombined[14], hence looking like a butterfly with wings spread. Combining all of the above makes the final DTFT vector. Of course, even further considerations can be used to speed up the algorithm or use less space, by using computer mathematical tricks such as bit (computer binary of 0 and 1) reversal and reordering. The FFT is an algorithm that naturally fits into computer algebra because of its system of binaries.

For those interested in DSP for sound engineering, in today's world of sound, we can record a waveform digitally (at some Hz of sampling - usually CD quality of 44.1 kHz, 128 bits), and view it graphically on screen in wave editing software such as Audacity or SoundForge, and Digital Audio Workstations (DAWs) such as Ableton Live and Cubase. Scientific analysis software such as Mathematica, Matlab, R and Python also have libraries which make this possible.

4.4 | Fastest Fourier Transform of the West!

The fastest algorithm[15] currently developed, implemented, publicly-known and free (Woot!) is the whimsically but very aptly named 'Fastest Fourier Transform of the West' (FFTW) by Matteo Frigo and Steven G. Johnson at the Massachusetts Institute of Technology, as a C/C++ library. Julia Language requires FFTW to be installed as a prerequisite.

4.5 | Industrial Uses of FFT

Using these software, we can mix any number of signals, waveforms or tracks to get a finished 'mix-down', introducing tweaks to each track (functions applied to the FT of the individual wave) to get a desired effect or sound. It is no wonder the sudden emergence and domination of urban-styled music like Hip-hop, D&B, Techno, their derivatives and remixes, and not to forget the remastering of older audio tracks (e.g. by Miles Davis or the Beatles) making them sound 'new' again with a seeming higher fidelity output despite the lower quality recording technologies and medium of the time. FFT is most definitely one of the core engines behind the voice-warping capabilities of modern karaoke machines.

The FFT algorithm makes all these possible, first by enabling view of the waveform in terms of frequency and spectrum, and furthermore enabling the concept of signal convolution amongst other filters. Another example is face recognition software embedded in Facebook and recent mobile devices. This can be achieved by combining Principle Components Analysis (Eigenvalue decomposition) together with Frequency analysis using FFT. Yet another use-case of FFT is the 'green screen' technology of mixing CG with real actors, and of course photo-editing. It is naturally used for cryptography and data compression as well.

[14] https://www.youtube.com/watch?v=nqaFs-msgUQ
[15] http://www.fftw.org/

5 | FFT FOR CONVOLUTION

We now come to the main topic of this paper where we discuss the use of FFT for convolutions of probability density functions (pdf). As we know, we can run into integration problems that are not trivial when we convolve random variables of arbitrary pdfs. What if we had a magic tool that can just take any shape and just numerically spit out the convolution as a vector of samples, which we could then interpolate from? But we do! It is the FFT and iFFT.

If we were to convolve a known function with another known function algebraically, we could obtain a very nice formula if we are lucky, but it only describes the convolution of these 2 functions - hence could be rather limiting - and what if we need to convolve 3 functions, the 3rd convolution may not have a trivial solution sometimes (e.g. try to convolve a Half-Normal distribution (a Normal distribution that is folded at the $y = 0$ point such that only positive values are considered, with twice Normal probability) 3 times. In other words - not all functions have a clear-cut method of generalising their convolutions.

5.1 | A Simple Example - 2 Discrete Signals

Let us start with an easy example - just a convolution of 2 discrete samples of signals - $A = \{2, -1, 3\}$ and $B = \{-15, 89, 0\}$. These signals are specified as integers in the real plane for ease, but the method encompasses any type of number, including complex numbers. First we show what is expected in a convolution of these 2. We can use a simple classroom example of using a table to compute the linear convolutions. As you can see, we extend each list of signals to be the same length of 4, and pad the non-specified signals with 0s. This is known as 'zero-padding'. We write the 1st list in the topmost row, and the 2nd list in the leftmost column, and we proceed to multiply element by element: [2 * -15], [-1 * -15] etc. Then, we sum the products diagonally up in a 45° angle - i.e. [-30], [178 + 15] etc. The final convolution output will be $\{-30, 193, -134, 267\}$.

Table linear convolution				
	2	**-1**	**3**	**0**
-15	-30	15	-45	0
89	178	-89	267	0
0	0	0	0	0
0	0	0	0	0

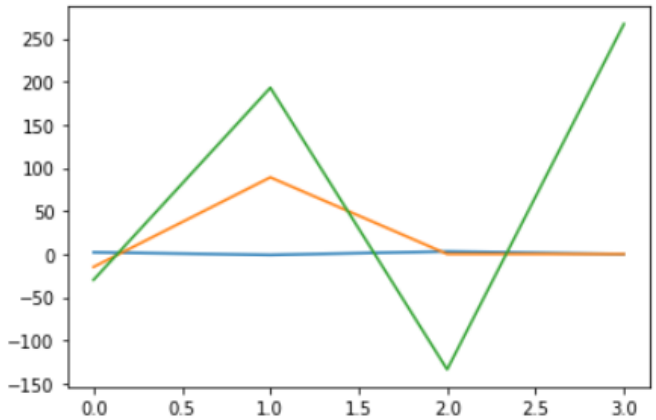
Here is some Python 3 code that does the above convolution using SciPy's FFT routine:

```
import numpy as np
from scipy.stats import norm
from scipy.fftpack import fft,ifft
import matplotlib.pyplot as plt
%matplotlib inline
```

```
discsample1=np.array([2,-1,3,0])
discsample2=np.array([-15,89,0,0])
dbin = 1 #because this is the scaling factor as per a count
##compute FFT
f_sample1 = fft(discsample1)*dbin
f_sample2 = fft(discsample2)*dbin
plt.plot(discsample1)
plt.plot(discsample2)
plt.show
#convolution of 2 discrete array of numbers
f_s1_con_s2 = f_sample1*f_sample2
#compute iFFT
s1_con_s2 = ifft(f_s1_con_s2)
plt.plot(np.real(s1_con_s2))
np.real(s1_con_s2)

```

```
array([ -30.,  193., -134.,  267.])
```



The input NumPy arrays were treated with the same 'zero-padding' before performing the FFT and convolution. Without additional 0s, the algorithm will wrap the non-zero samples at the end of the vector to the front, hence causing some over-convolution, which is definitely not the intended output. One could think of zero padding as giving the signals enough room to

fold over each other without excess overlap. In the case of these short arrays of signals, not much padding is required.

From the Python code, it is clear the logic of the computations. We ready the samples with zero-padding, then we compute their FFTs. We combine them by element multiplication, and we compute their iFFT to receive the final convolved product!

Easy right? This logic can be applied to any arbitrary signal of arbitrary shape, and some of the more useful intents mentioned in the more mathematical sections earlier is the use of FFT for convolving probability density functions. Mathematicians and Physicists use it as a way to prove their theorems, discover more advanced shortcuts, analyse relationships between functions, and of course to develop commercial solutions with FFT as the underlying algorithmic engine.

A brief recap on Fourier transform as a convolution engine:
If f and g are piecewise continuous, bounded and absolutely integrable on the z-axis then:

$$\mathcal{F}\{(f \circledast g)(z)\} = \sqrt{2\pi}\mathcal{F}\{f(z)\}\mathcal{F}\{g(z)\}$$

and inverse Fourier transform:

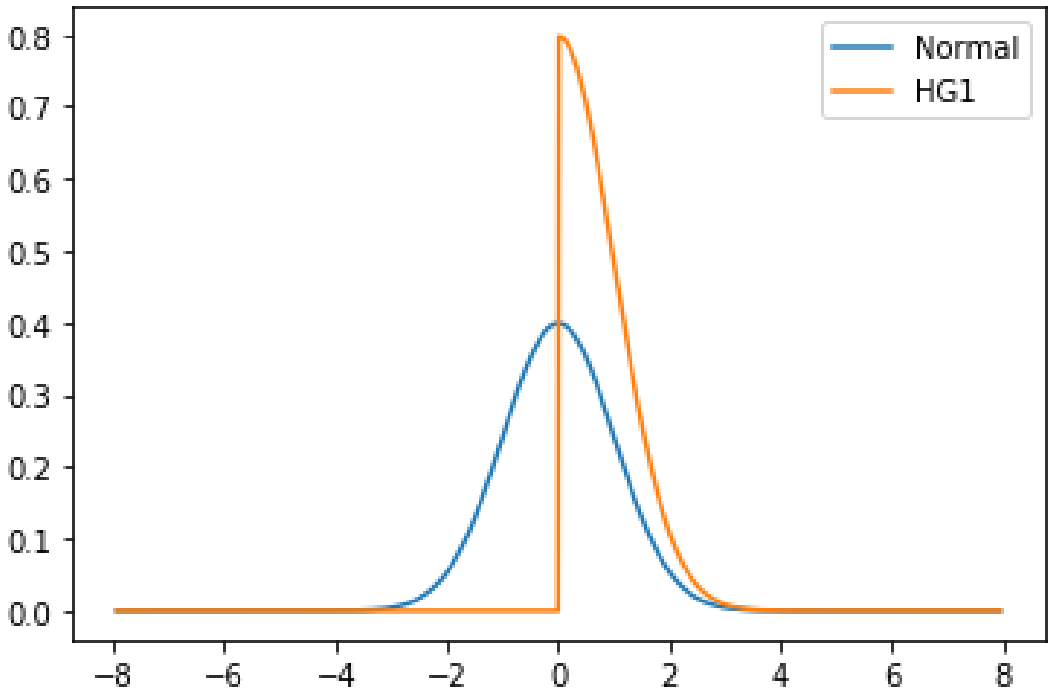
$$\mathcal{F}^{-1}\{\mathcal{F}\{f(z)\}\mathcal{F}\{g(z)\}\} = \frac{1}{\sqrt{2\pi}}(f \circledast g)(z)$$

Note that most implementations of FFT, despite the formal definition of FT for convolution above, do not implement any $\sqrt{2\pi}$ scaling, as other types of scaling for other purposes are possible - like our use of $dBin = 1$ in the Python code.

5.2 | 2 Half-Gaussian Signals

A pdf can be considered as yet another function which specifies a shape. Hence it can be conveniently treated as a signal as well. We shall feature Half-Gaussian convolution rather than boring Gaussians because it accentuates the point of using FFT as the convolution engine. It shall become clear soon enough.

Firstly, what is a Half-Gaussian or Half-Normal[16]? It is a special case of the Folded Normal distribution, 'folded-over' at the mean of 0.



Let $A \sim N(0, 1)$ and $Y = \|A\|$. This means, Y consists of only absolute values of A - no negatives. This implies the probability

[16]https://en.wikipedia.org/wiki/Half-normal_distribution

of positive Y is twice that of positive A and 0 in the negative plane. Basically A's negative values have been 'folded-over' to double up in the positive plane. The HG pdf starts with the mode at 0 and has a Normal looking tail, it however has a new mean, implying it has now become a distribution with skew and excess kurtosis, compared to the Normal curve. If $A \sim N(0, \sigma^2)$, then it has a Normal distribution with the mode and mean 0 and variance σ^2. $Y \sim HG(\sigma^2)$ as a Half Normal carries twice the positive weightage but resides only in the positive plane. This implies that the convolution is not a simple sum of means and variances unlike its mother, the Normal. The pdf:

$$f(y) = \frac{dCDF_{HG}(y)}{dy} = \frac{2}{\sigma\sqrt{2\pi}} e^{-\frac{(y-\mu)^2}{2\sigma^2}} \qquad 0 < y < \infty.$$

So now the question - what is the pdf of $X + Y$ if X also follows the same distribution but uncorrelated to Y? Naturally the answer is the convolution of both. We seek pdf of Z where $Z = X + Y$, with f and g being Half-Normal pdfs:

$$(f \circledast g) = \int_{-\infty}^{\infty} f(z-x)g(x)\,dx$$

$$= \mathcal{F}^{-1}\{(\phi(t))^2\}(y)$$

where $\phi(t)$ is the characteristic function aka FT of the HG pdf.

$$= E(e^{itY}) = \int_{-\infty}^{\infty} f(y)e^{ity}dy$$

After trolling through the algebra and calculus, we provide the double Half-Gauss pdf as:

$$HG_2 = \frac{2\sigma}{\sqrt{\pi}} e^{-\frac{z^2}{4\sigma^2}} \mathrm{erf}(\frac{z}{2\sigma}) \qquad 0 < z < \infty$$

where erf is the error function that is highly related to the cumulative Normal function.

$$\mathrm{erf}(z_2) = \frac{2}{\sqrt{\pi}} \int_0^{z_2} e^{-r^2}\,dr$$

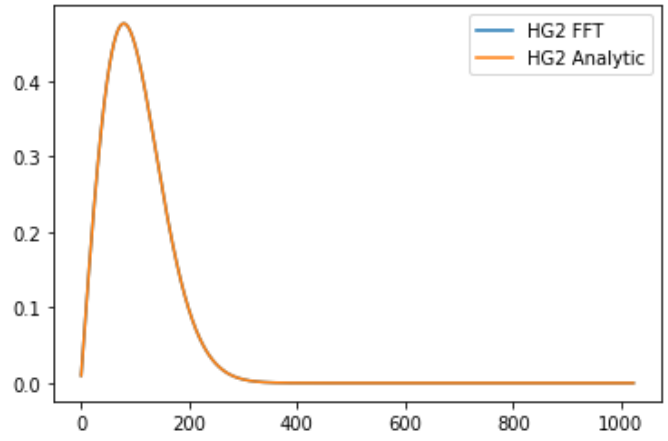
```
import numpy as np
from scipy import special
from scipy.special import erf
from scipy.stats import norm
from scipy.fftpack import fft,ifft
import matplotlib.pyplot as plt
%matplotlib inline
```

```
def hg2conv(x, sigma):

    pi = np.pi
    #must use np.array to get back in correct shape, and is to remove neg side
    x = np.array([max(num,0.0) for num in x])
    errf = erf(x * 0.5 / sigma)
    ans = (2 / np.sqrt(pi) / sigma) * np.exp(-(x ** 2 / (4 * sigma ** 2))) * errf

    return ans
```

This is where we start to observe that there can be difficulties trolling through the math for a 3rd, 4th and more convolution of Half-Gaussian pdfs, as the pdf function does not reveal a simple form to accomodate a new pdf naturally, unlike Normal, Student T or Gamma distributions. A solution could be to substitute the erf function with an expansion or approximation such that the integration is easier to continue with. But why not just wave our magic FFT wand? Abracadabra!

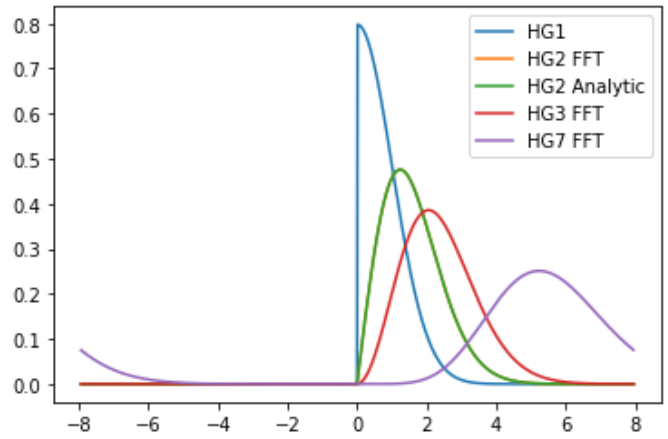


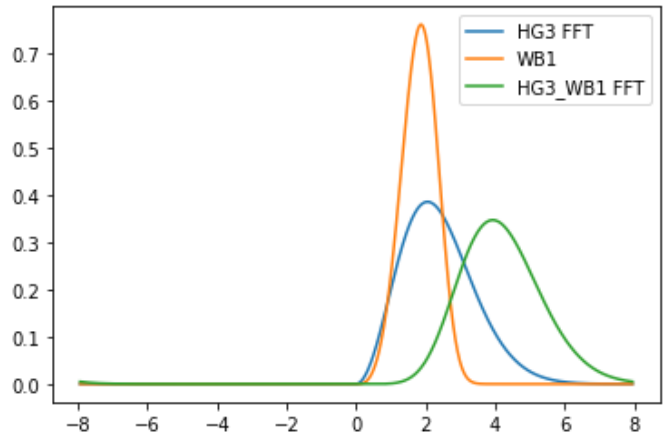

```
N = 1024
a=10**(-15)
end=norm(0,1).ppf(a)

sample=np.linspace(end,-end,N)
dbin = sample[1]-sample[0]
pdfnorm=norm(0,1).pdf(sample)
hgpdf=pdfnorm + pdfnorm #'create 2x normal distribution'
hgpdf[0:int(N/2)] = 0.0 #'remove the left side of the normal tail'
f_pdf = fft(hgpdf) * dbin
f2_pdf = f_pdf * f_pdf
hg2pdf = ifft(f2_pdf)/dbin
r_pdf = np.real(hg2pdf)
```

```
hg2pdf_an = hg2conv(sample, 1)
plt.plot(np.real(hg2pdf), label="HG2 FFT")
plt.plot(r_pdf, label="HG2 Analytic")
plt.show
plt.legend()
```

Look at the graphical comparison (the x-axis labels are counts of values - to 1024) - we only see 1 curve, meaning the values are pretty much the same! Notice that we had not explicitly zero-padded this example except removing the 512 values in the negative plane. This only works for this HG example as when we removed the left half of the signal, it is implicitly padding $2N$ 0s with $N = 512$. Note that zero-padding, to be on the safe side requires to be at least 2x the sample size. $dbin$ is the same scaling used on all 3, which really is the equi-spacing of the x-axis of the distribution whose lower and upper boundaries we chose to be the near 0 probability limits of the Normal distribution (say around 7 standard deviations away from the mean).






```
f3_pdf = f2_pdf * f_pdf
f7_pdf = f_pdf**7
hg3pdf = ifft(f3_pdf)/dbin
hg7pdf = ifft(f7_pdf)/dbin
sample2 = sample
plt.plot(sample2, (hgpdf), label="HG1")
plt.plot(sample2, np.real(np.fft.fftshift(hg2pdf)), label="HG2 FFT")
plt.plot(sample2, np.fft.fftshift(r_pdf), label="HG2 Analytic")
plt.plot(sample2,np.real(hg3pdf), label="HG3 FFT")
plt.plot(sample2,np.real(hg7pdf), label="HG7 FFT")
plt.show
plt.legend()
```

```
def weibull(x,n,a):
    return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
```

```
wb1 = np.zeros(N)
wb1[int(N/2) :]=weibull(sample[int(N/2) :],2,4)
f_wb1 = fft(wb1) * dbin
f_hg3_wb1 = f3_pdf * f_wb1
hg3_wb1pdf = ifft(f_hg3_wb1)/dbin
plt.plot(sample2,np.real(hg3pdf), label="HG3 FFT")
plt.plot(sample, wb1, label="WB1")
plt.plot(sample, np.real(np.fft.fftshift(hg3_wb1pdf)), label="HG3_WB1 FFT")
plt.show
plt.legend()
```

To conclude this portion, we show here that it is trivial to convolve 3 or even 7 HG distributions - however we purposely show the 'wrap-around' effect of 7 - whose right-hand tail values suddenly appear in the negative plane, clearly erroneously, and reiterate on the importance of proper 'zero-padding'.

One would notice that we applied an fftshift function to our output data for HG2. This is because within most FFT implementations, the information with regard to the x-axis is lost - replaced by an automatic enumeration of each signal in the sample. Due to even periodicity of this FT, it requires a realignment back to the origin - i.e. it shifts FFT output such that the 0 frequency component is back at the centre. If the periodicity is odd, we do not require the fftshift. Look! Even convolving the HG3 with a Weibull distribution $wb(x) = \frac{\alpha}{\lambda}(\frac{x}{\lambda})^{\alpha-1}e^{-(\frac{x}{\lambda})^{\alpha}}$ is trivial!. Isn't this truly magical?

5.3 | Black-Scholes & General Option Pricing

Exciting as this may already be, being able to magically combine arbitrary distributions just by regarding their shape as samples of signals, we welcome the reader to witness the grand event of our magic show. We shall use FFT convolution as our engine to price - not one, but many European vanilla options (single strike date at option maturity) simultaneously, all the while preserving the fundamental pay-off function and probability kernel with minimal pre-transformation (most papers you will find on it involve pre-solving the characteristic functions). While it may not be the most optimised nor accurate (well actually...) formulation, it is meant to showcase the simplicity of it all. For a more formal and rigorous approach, we refer the reader to Carr and Madan[17].

When one refers to a textbook on option pricing, she/he often gets overwhelmed by a whole thesaurus of jargon (probably concocted to keep the sciences relevant and force the majority of humans to stick to their jobs whilst sounding like an expert of the field) like 'risk-neutral', 'numeraire', 'arbitrage-free', 'delta-neutral portfolio' yada yada, and any deviation seems to be met often with awkward stares and intellectual verbal challenges. We digress for a little financial-industry sarcasm.

Today, we will put aside all of that and approach Option pricing as simple convolution of the 2 original signals - the payoff and the pdf, just like the MC or Tree approaches. For this example, we will be using a Black-Scholes variant known as Garman-Kohlhagan (GK) which has the same formula as a Black76[18] Forward Option pricer.

GK is formulated primarily for pricing Foreign Exchange vanilla options (FXO). Recap that an option is the right not obligation to purchase (call) or sell (put) an underlying asset at a certain pre-determined exercise/strike price at a pre-determined future date, for an upfront smaller amount known as an option premium (which is the output of the GK model).

Not getting into too much detail but for FXO, there are generally many industry-conventions to abide by, which complicate the pricing of an option, e.g. a call option on say USD/MYR (USD call - MYR put) normally demands a premium in terms of the stronger, more liquid currency - i.e. USD (base / left-hand side) (at least for now...), but running GK on it presents a theoretical premium output in terms of the terms (right-hand side) currency - i.e. MYR. Also not considered here is the basic industry standard of considering implied or local volatility surfaces - although that is a relatively simple extension - simply requiring a bit of added pre-working on the volatility parameter and reading the input into the algorithm as a function rather than a constant.

Hence, for industrial use, more transformations and scaling will be required, but for our FFT example, we are simply going to match this GK premium in terms currency (MYR) output using FFT. First, to recap the GK formula:
S = Spot FX rate
K = Strike rate
r_d = Domestic interest rate
r_f = Foreign interest rate
T_d = Time to option delivery (the time applied to the interest rates)
T_e = Time to option expiry (the time applied to the volatility)
σ = Volatility of the option
$\epsilon = 1$ for Call, -1 for Put

$$GK(S, K, r_d, r_f, T_d, T_e, \sigma, \epsilon) = \epsilon(Se^{-r_f T_d} N(\epsilon d_1) - Ke^{-r_d T_d} N(\epsilon d_2))$$

$$d_1 = \frac{ln(\frac{S}{K}) + (r_d - r_f)T_d + \frac{\sigma^2 T_e}{2}}{\sigma\sqrt{T_e}}$$

$$d_2 = d_1 - \sigma\sqrt{T_e}$$

This is the celebrated standard Black76 formula. We will reformulate it as an integral in terms of the standard Normal pdf that can be integrated in standard $\int_{-\infty}^{\infty} xp(x)dx$ fashion.

$$\hat{\sigma} = \sqrt{\sigma^2 T_e}$$

[17] https://engineering.nyu.edu/sites/default/files/2018-08/CarrMadan2_0.pdf
[18] https://www.lme.com/en-GB/Trading/Contract-types/Options/Black-Scholes-76-Formula

where $\hat{\sigma}$ is the standard deviation of an option (not to be confused with volatility)

$$\hat{\mu} = (r_d - r_f)T_d - \frac{\hat{\sigma}^2}{2}$$

where $\hat{\mu}$ is the drift or mean of an option which takes into account Ito's Lemma correction to the drift.

$$p(x) = \frac{e^{\frac{-x^2}{2}}}{\sqrt{2\pi}}$$

i.e. the standard Normal pdf.

$$DF_{r_d} = e^{-r_d T_d}$$

i.e. domestic discount factor

$$\boxed{GK = DF_{r_d} \int_{-\infty}^{\infty} max(\epsilon(Se^{\hat{\mu}+\hat{\sigma}x} - K), 0)p(x)dx} \quad (1)$$

implying that we can get rid of any max or Heaviside functions by changing the integral bounds. Note that for FFT pricing, we will only need this main GK equation (1) above. The rest that follows below is simply to enlighten the reader and provide some intuition using a numerical or direct integration formulation.

$$LB_{Call} = UB_{Put} = \frac{ln(\frac{K}{S}) - \hat{\mu}}{\hat{\sigma}} = -d_1$$

$$GK_{Call} = DF_{r_d} \int_{LB_{Call}}^{\infty} (Se^{\hat{\mu}+\hat{\sigma}x} - K)p(x)dx$$

$$GK_{Put} = DF_{r_d} \int_{-\infty}^{UB_{Put}} -(Se^{\hat{\mu}+\hat{\sigma}x} - K)p(x)dx$$

which can of course also be solved with quadrature methods like Gauss-Legendre, with $\{-\infty, \infty\} \approx \{-7, 7\}$ since those points are near 0 probability under a standard Normal.

Just to put this in layman terms, for say a Call, we are integrating from the starting lower bound $x = -d_1$ where the spot price in the future has value (i.e. $S_t(x) > K$) to ∞, all the possible future spreads of the spot price against strike rate, weighed by the Normal probability function. Then we present-value that future-valued integral by the domestic discount factor. Everything $x \leq -d_1$ has 0 value, hence not worth anything.

Now, we return to FFT. Recall the standard formula for convolutions:

$$h(z) = \int_{-\infty}^{\infty} f(t)g(z-t)dt$$

and that FFT does the same thing (to some extent depending on how it is specified), it implies that we can choose one of the functions, $f(t)$ say to be the payoff function and $g(t)$ to be the Normal pdf - it does not matter which one, and if we are able to convolve the functions, $h(z)$ will become the new GK pdf, describing theoretically, all the GK values in the space, with 1 of the results being the option value we seek! Now isn't that awesome, not only are we integrating the payoff function with the density kernel, we are also receiving all permutations of the payoff function combined with density! Now that's some powerful magic right there.

So what we mean by FFT doing this to some extent is that it provides in just a single run, a range of GK premiums and then tapers off, because it only goes as far as the range of S_ts we specify as a signal to convolve, and then starts to fall quickly to 0 (as it would have convolved with 0 or non-specified values). Not to fret though, as the point where it tapers off, is many option standard deviations ($\hat{\sigma}$) away, so in real practice, option pricing never usually reaches that far. You will see this in a minute. We begin by giving Python code for analytic GK:

```
def gk(S, K, rD, rF, Td, Te, sigma, icp):
    #Garman Kohlhagen (Black76) price for FXO Vanillas
    #S: spot price
    #K: strike price
    #Td: time to maturity (applied to forward rates)
    #Te: time to expiry (applied to volatility)
    #rD: domestic interest rate
    #rF: foreign interest rate
    #sigma: volatility of underlying asset
    #icp: 1 for call, -1 for put

    rft = rF * Td
    rdt = rD * Td
    sigsqrt = sigma * np.sqrt(Te)
    mu = rdt - rft + sigsqrt * sigsqrt * 0.5
    DF_rd = np.exp(-rdt)
    DF_rf = np.exp(-rft)

    d1 = (np.log(S / K) + mu) / sigsqrt
    d2 = d1 - sigsqrt

    optval = icp * (S * DF_rf * norm.cdf(icp * d1, 0.0, 1.0) - K * DF_rd * norm.cdf(icp * d2, 0.0, 1.0))

    return optval
```

and the FFT version:

```
#Garman Kohlhagan FFT using convolution
#Generate normal distribution pdf as a signal
#first we introduce the call option details and pay off function
S = 2.38; K = 2.665; rd = 0.09; rf = 0.05; Td = 0.56; Te = 0.56; Vol = 0.3; iCP = 1; N_samp = 8192;
#we now compute mu and sigma as per parameters to a normal distribution
Sig = np.sqrt(Vol * Vol * Te)
#mu must contain the Ito Lemma drift correction, and the time.
Mu = (rd - rf) * Td - 0.5 * Sig * Sig
#d1 = (np.log(S / K) + Mu) / Sig
#d2 = d1 - Sig
#-d1 is lb_call and ub_put, where St = K so no value
a=10**(-15) #an extreme range
end=norm(0,1).ppf(a) #get the cumulative normal inverse of that extreme left percentile
#build a equispaced vector that goes from lower extreme to upper - Normal is symmetric
sample=np.linspace(end,-end, N_samp+1) #sample bounds
dbin = (-end -end) / N_samp #the bin scaling for FFT and iFFT
normsample1=norm(0,1).pdf(sample) #generate the pseudo-full N(0,1) pdf
#payoff function
callsample2=[max(iCP * (S * np.exp(Mu + Sig * n) - K), 0)  for n in sample]
#zero padding
zp = np.zeros(N_samp * 2) #2n
normsample1 = np.concatenate((normsample1, zp), axis=None)
callsample2 = np.concatenate((callsample2, zp), axis=None)
```

```
##compute FFT and scale
f_sample1 = fft(normsample1)*dbin
f_sample2 = fft(callsample2)*dbin
#plt.plot(normsample1)
#plt.plot(callsample2)
#convolution by element-wise multiplication
f_s1_con_s2 = f_sample1*f_sample2
#compute iFFT and scale
s1_con_s2 = ifft(f_s1_con_s2)/dbin
plt.plot(np.real(s1_con_s2))
plt.show
```

As can be seen in our Python FFT GK implementation, we are going to feature an example with:

$$\{S, K, r_d, r_f, T_d, T_e, \sigma, \epsilon\} = \{2.38, 2.665, 9\%, 5\%, 0.56, 0.56, 30\%, 1\}$$

with sample size N for FFT of 8192. Note that in general FFT, to optimise the use of the engine due to it's binary properties,

we want to specify N in order of some 2^n samples, although it would work with any number really. The higher N is, the more accurate the results will get.

Here is our output comparison:

```
#we present value the vector with domestic DF
GK_FFT = np.real(s1_con_s2)*np.exp(-rd * Td)
#and we pick the option value at last index we specified our sample for
o_gk_fft = GK_FFT[N_samp]; o_gk_an = gk(S,K,rd,rf,Td,Te,Vol,iCP);
absdif = o_gk_fft - o_gk_an
reldif = absdif / o_gk_an
print("FFT: ",o_gk_fft) #is our option value
print ("GK Analytical: ",o_gk_an) #garman-kohlhagan value
print ("Abs Diff / Rel Diff: ", absdif,"   ", reldif)
```

```
FFT:  0.12415169648939141
GK Analytical:  0.12415168257463294
Abs Diff / Rel Diff:  1.3914758470456867e-08     1.1207869423833306e-07
```

Our FFT approach got us the option value exact to 7 decimal places - which is pretty good - Hooray! Other numerical approaches such as Monte Carlo will likely still not converge to that type of accuracy with only 8K runs. In practice though, as option pricing is already just an approximation (due to market changes not really being Normally distributed), this type of accuracy is multiple-folds more than enough, unlike in other fields - say astrophysics, or audio-visual engineering where the demand for accuracy may be higher, of which to achieve, we simply increase the sample size. On that note we digress to inform that we already meet the exact GK analytical option value consistently (across widely different tuples of inputs) to 6 decimal places with $N = 1024$.

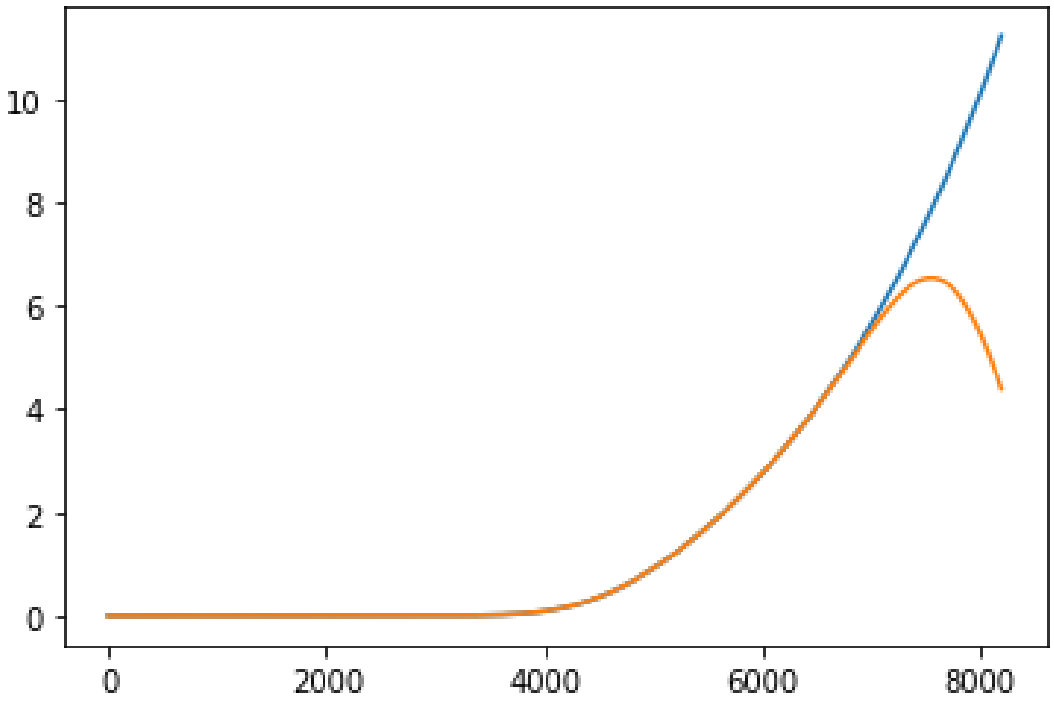
If we scrutinise the FFT implementation, notice that what we are doing is simply treating the Normal pdf as a signal, and the payoff-function as another signal, (as aforementioned - only using the main integral equation (1)) with both tied together by the same Normal percentile which $\sim U(0, 1)$ meaning they are both tied to the same equi-distanced Uniformly distributed variable, and that implies there is correlation between the 2 functions. Nevertheless, FFT took it all in and handled it to perfection.

```
#we know Spot is priced at N_samp, so now we build around it using +/- Sig * n
s0_sample = np.double([S * np.exp(Sig * n)  for n in sample])
#and we build a vector of GK analytic premiums to compare with FFT
gks0 = gk(s0_sample, K, rd, rf, Td, Te, Vol, iCP)
#FFT range of N_samp/2 to N_samp/2 + size of gks0 values are equivalent to those in gks0,
#but taper off somewhere halfway - but this is many standard deviations away
lb = int(N_samp/2); ub = lb+len(gks0)
GK_FFT2 = GK_FFT[lb:ub]
```

```
plt.plot(gks0)
plt.plot(GK_FFT[int(lb):int(ub)])
plt.show
```

The beauty of FFT is that it does not just produce that 1 premium for that 1 $\{S, K\}$ pair, unlike the analytical GK formula. Simply specifying that pair in FFT already generates us multiple premiums across the space! To show this, we will compare the other values that FFT generated, however we first have to align the values.

Here, for our comparison purposes we first build a vector of all the possible Spot prices based on the multiplier on $\hat{\sigma}$ and generate the entire vector of analytical GK, keeping the rest of parameters the same. And, for our FFT vector, we copy starting from a lower bound of $N/2$ to upper bound of $N/2 + sizeof(GK)$. This will align perfectly both of the output, such that we can see exactly when FFT will diverge. So now, we plot to explain the range of values that FFT will be valid for:



Visually, it can be observed that we matched closely about 7000 (where the price starts to taper off) GK option premiums in just 1 run. If we have a difference threshold in mind as to how much we really need to match the analytical formula we can do so with some code:

```
d = np.abs(GK_FFT2 - gks0)
us=np.where(d>1e-6) #find us the index where divergence > threshold
divergence = us[0][0]
print ("Acceptable range (based on threshold) is up to: ", divergence -1)
print ("Option Value: ", GK_FFT2[lb], gks0[lb])
print ("Option Value: @ 350 points before: ", GK_FFT2[lb-350], gks0[lb-350])
print ("Option Value: @ 14 points after: ", GK_FFT2[lb+14], gks0[lb+14])
print ("Last Option Value point before divergence: ", GK_FFT2[divergence - 1], gks0[divergence - 1])
print ("Divergence > Threshold of 1e-6", GK_FFT2[divergence], gks0[divergence])
print ("Spot before and at Divergence point", s0_sample[divergence-1],s0_sample[divergence])
print ("Usable range (based on threshold) is up to (x) stdevs away: ", (s0_sample[divergence-1] - S)/Sig)
```

```
Acceptable range (based on threshold) is up to:  5497
Option Value:  0.12415169648939141 0.12415168257463294
Option Value: @ 350 points before:  0.035628588710849333 0.03562858093247506
Option Value: @ 14 points after:  0.12967460966475908 0.12967459555825223
Last Option Value point before divergence:  1.7270085375791047 1.727009527661259
Divergence > Threshold of 1e-6 1.7288482853196296 1.7288492858684377
Spot before and at Divergence point 4.37935149118082 4.381258065436086
Usable range (based on threshold) is up to (x) stdevs away:  8.905819375874062
```

So here, we compute a few comparison metrics to show the reader that everywhere before the actual option premium based on S, for a Call, is pretty much on the dot. We set a difference threshold of 1e-6 (which is really tiny) and find the index where the deviation is more than the threshold for this case to be at 5498, hence based on this very rigid threshold, we already accept 5497 premiums!

In reality though, this Spot at the 5497th point is already at about $8.9\hat{\sigma}$ away! For general financial industry, this is ideal as there is no way that a market would be pricing options that far away - and if there is such an attribute - GK would not be the correct model to use in any case.

In general, using FFT to calculate a simple European option value may be an overkill, however one ought to consider the limitless expansion possibilities. For one, here in our example we have displayed that our approach generates all the option values for a range of Spots, but it is another way to say that it generates all the option values for a range of Strikes - which we can use a simple interpolation scheme (usually linear) to obtain extremely good approximations for any strike rate within the range.

Using FFT and its convenient convolution capabilities as the pricing engine, we can obviously take non-Normal kernels as our pdfs, however the Ito lemma correction to the drift must be properly worked out as non-symmetric pdfs may require us to take higher Taylor series expansion orders on T_e. Pricing exotics such as Basket options is a natural regard of convolution.

Real-word option pricing is a vast and complex subject matter that can comprise of any type of payoff structure imaginable including path-dependency, copulae etc. We do not know yet all the answers as to how FFT could be used in all situations, but even more reason to pick it up, begin researching, using and developing the FFT as a possible pricing engine. Moreover, it is a fun tool to play around with, no doubt a lot of thought and effort (and experimentation) will have to go into it. Fun fact - did you know that it is faster to derive a large correlation vector using FFT rather than compute the Pearson coefficients linearly - we leave this as a fun topic to research for yourself.

6 | FINALLY

We hope that by providing a small insight into some little facts-and-fun, and a demonstration of its implementation for a bit of application can help put a bit more spotlight on FFT as an engine, for those recently discovering it, and those who studied and forgot, to build an appreciation towards its history, its contribution, the methods and journey to develop it, to its current use. Next time if you turn on the TV, or send a Whatsapp, think out loud ,"Thank Gauss! Oh and you too Bernoulli, Cotes, Euler, Fourier, Tukey and Cooley ;-)... et al!"

The author generally likes the notion that the currently most versatile and quickest FT machine on this Earth is the very organic, very flaw-ful, yet incredibly capable with unlimiting potential human brain. Think about it - we can perceive and convolve in our heads, a whole lot of signals simultaneously. We can concurrently smell, taste, drive (although highly not advisable), plan our day and think of not knocking the car in front, while talking on the phone (seriously - don't), and knowing how hot the steering wheel feels under the sun. All sorts of processing, mixing, interpolating, extrapolating, all at the seeming same time and almost instantaneously. He hopes to witness some day the emergence of a super (but nano) computer that is totally multidimensional FFT hardwired, with complete FFT-adopted architecture directly within all its chipset and operating system (probably a Linux distribution will be the quickest and most versatile choice to adopt this). It can be used to power all kinds of even faster development and industries. Combined with another piece of math wizardry known as Quaternions, an extension to the complex number system but with 3, not just 1 $\sqrt{-1}$ terms, much more capable than the Euler rotation to describe spacial rotation or morphing, an FFT-based system can generate virtual worlds for VR games and movies, and well who knows - a whole digital world with interaction capability. Just for interest, Quaternion math was developed by Irish mathematician William Rowan Hamilton in 1843. Some extremely interesting intuition-building videos are available on Youtube on this topic[19].

A shout out to all the medical people in the world for taking care of the weakened and desperate during this dreadful Covid19. I wish all the best to the world! Let's focus on development - and if we run out of space, inner development - but please not war!

Funding: This paper was completely researched and self-funded by the author - Gene Boo.

Conflict of Interest: The author declares that there is no conflict of interest.

[19] https://www.youtube.com/watch?v=d4EgbgTm0Bg

6.1 | Bibliography

References

1. Kreysig E. Advanced Engineering Mathematics, 8th Ed.. 1999.
2. James Cooley PW. Historical Notes on the Fast Fourier Transform. 1967.
3. Peter Carr DM. Option Valuation Using the Fast Fourier Transform. 1999.
4. Hilpisch Y. Derivatives Analytics with Python. 2015.
5. Milton Abramowitz IS. Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables. 1964.
6. Cornelis Osterlee LG. Mathematical Modeling and Computation in Finance: With Exercises and Python and Matlab Computer Codes. 2019.

How to cite this article: Gene B. (2020), Fast Fourier Transform as an Engine for the Convolution of PDFs and Option Pricing Using Python