This section briefly discusses the fundamentals of the CKKS HE scheme, as described in the original paper by Cheon et al. [CKKS], and how it is implemented in this project. Also, to provide further insights and an investigation in specialising HE schemes, a bespoke CKKS implementation is detailed, called MeKKS.

# CKKS Primitives

Fundamentally, the CKKS scheme encrypts plaintext polynomials into ciphertext polynomials. Addition and multiplication operations can then be performed on the data, introducing uncertainty. Ciphertext polynomials can be decrypted to retrieve approximations of plaintext polynomials, with the accuracy depending on the number of operations performed.

To encrypt vectors of real values, they must first be encoded as polynomials in the ring $\mathcal{R} = \mathbb{Z}[X] / (X^N + 1)$, where $N$ is a power of 2. During encoding, the real values must be rounded so precision is preserved by multiplying by a \textit{scaling factor}, $\Delta$. Once the vector has been encoded, it can be encrypted into a \textit{pair} of polynomials: the ciphertext.

Consider two vectors $\vec{v}$ and $\vec{w}$. To encode these vectors, they are scaled to $\Delta \vec{v}$ and $\Delta \vec{w}$. They can then be encrypted and multiplied to give ciphertext equivalents of $\Delta^2 (\vec{v} \bigodot \vec{w})$. This implies that a sequence of multiplications will increase the scale factor indefinitely. To overcome this, CKKS introduces a \textit{rescaling} procedure, which can be understood as dividing the ciphertext by $\Delta$ to reduce the scale.

However, rescaling is not a free operation, so it cannot be applied to allow unlimited multiplications. CKKS is a levelled HE scheme, so each ciphertext resides at a discrete level. Each level has a coefficient modulus $q$ that dictates a ciphertext’s coefficients must be in $\mathbb{Z}\_q$. When a polynomial is first encrypted, it exists in the maximum level, $L$, with coefficient modulus $Q = q\_0 \dot \Delta^L$, for a \textit{base modulus} $q\_0$. When a ciphertext is rescaled, the coefficient modulus is divided by $\Delta$, reducing it to $q\_0 \cdot \Delta^{L-1}$. Hence, the ciphertext is lowered to the next level. If a ciphertext reaches level 0, no further multiplications can be applied to preserve the encrypted values. For correct decryption, the coefficients of a polynomial cannot exceed $q\_0$. Consequently, in practice, $q\_0$ is much larger than $\Delta$.

## Encoding and Decoding

The plaintext message space is the \textit{cyclotomic polynomial ring} $\mathcal{R} = \mathbb{Z}[X] / \Phi\_M(X)$, where $\Phi\_M(X)$ is the $M$-th cyclotomic polynomial and $M$ is a power of two. Encoding is the process of mapping a complex vector to an element in $\mathcal{R}$, and decoding is the process of reversing it.

Another way of defining decoding is as a mapping of an element $r \in \mathcal{R}$ to a vector in $\mathbb{C}^N$ using the embedding $\sigma : \frac{\mathbb{R}[X]}{X^N + 1} \rightarrow \mathbb{C}^N$. This is applied to each root of $\Phi\_M(X)$ through the evaluation of $r$ as

EQUATION

where $\xi = e^{\frac{2 \pi i}{M}}$ is a primitive $M$-th root of unity.

However, to establish a one-to-one mapping, the input must be \textit{scaled} and \textit{rounded} during encoding, and half of the complex vector must be \textit{discarded} during decoding.

As a result of the above, the CKKS encoding function has the form given in Equation \ref{eq:encode}.

## Operations

A full list of the operations supported by CKKS and their definitions is included in Appendix \ref{app:operations}.

## Rescaling

The rescaling function introduced above is defined by equation \ref{eq:rescale}.

EQUATION

Rescaling truncates some of the least-significant bits of a ciphertext by dividing by a power of the scaling factor. Practically, this is performed after every multiplication to scale $\Delta^2$ down to $\Delta$.

## Rotations

An advantage of RLWE based HE schemes over others is the ability to \textit{rotate} ciphertext slots. Given a vector $\vec{V} = [v\_0, v\_1, v\_2, \ldots, v\_n]$ and an offset $d$. A rotation would cyclically shift each element by $d$ indices\footnote{For example, given $\vec{v} = [1, 2, 3, 4, 5]$ and $d = 2$ would produce $\vec{v}’ = [4, 5, 1, 2, 3]$.}. This is achieved by homomorphically performing a \textit{Galois automorphism} on the ciphertext using standard results from \textit{Galois theory}. However, these operations are significantly the most expensive operations offered by RLWE-based schemes [ROTATIONBAD].

## RNS Optimisation

SEAL utilises a \textit{*residue number system*} (RNS) to overcome the slow computations caused by very large polynomial coefficients requiring arbitrary precision arithmetic [RNS]. This exploits the Chinese Remainder Theorem (CRT) to decompose the coefficients in $\mathbb{Z}\_q$ into $n$ smaller ones: $\mathbb{Z}\_{p\_1}, \ldots, \mathbb{Z}\_{p\_n}$. To do this, the CRT uses a \textit{*moduli switching chain*} $p\_1, \ldots, p\_n$ such that $\prod\_i p\_i = q$ and each $p\_i$ is a prime number requiring fewer than 64-bits to store. Thanks to hardware limitations, it is faster to compute the $n$ separate multiplications in $p\_1, \ldots, p\_n$ than it is to compute a single multiplication in $q$. The results of the separate multiplications can be easily combined thanks to the isomorphism between $\mathbb{Z}\_q$ and $\mathbb{Z}\_{p\_1}, \ldots, \mathbb{Z}\_{p\_n}$.

Unfortunately, as a result of the RNS optimisation, $q\_l = q\_0 \Delta^l$ cannot always be maintained due to the difficulty of finding every $q\_l$ as the product of $n$ primes. To overcome this, SEAL instead defines $q\_l = q\_0 \prod\_{i=1}^l p\_i$ where $q\_0$ must be prime. Where the original CKKS scheme divides by $\Delta$ to rescale, SEAL divides by $p\_l$. Consequently, the resulting scaling factor is only approximately equal to $\Delta$. Therefore, in practice, the scaling factor must be manually reset to $\Delta$ after each multiplication and rescaling.

# MeKKS

Initially, the goal of implementing the CKKS scheme from first principles, named MeKKS, was to understand its fundamental mathematical principles further. There were little to no expectations of providing a performance improvement. However, during implementation, it became apparent that there were advantages to a specialised HE scheme that can overcome weaknesses found in SEAL.

## Limitations

The primary limitation of MeKKS was the language used for implementation. Traditionally, encryption schemes are written in C or C++ because they are low-level, so usually run faster. Consequently, the more computationally expensive operations (such as the multiplications between large prime numbers required by CKKS) would run much quicker than in Python.

Another limitation was the number of optimisations that could be applied. Since the focus of the implementation was on understanding, the implementation began with the foundational CKKS scheme. However, since this scheme was released, many optimisations have been produced. For example, using residue number systems, bootstrapping extensions, or reduced approximation error [RNS, BOOTSTRAPPINGHEAAN, RAE]. The opportunities for developing MeKKS are extensive. However, it was essential to define a clear endpoint for the implementation to schedule the project. Therefore, the performance expectations compared to SEAL were significantly bounded.

## Implementation

The first iteration of MeKKS implementation began using the scheme outlined by Cheon et al. in 2016 [CKKS]. This allowed all of the functionality described in §\ref{sec:CKKS} to be implemented following the SEAL API, allowing for easy integration with the existing application.

However, the performance of this implementation meant it was infeasible to integrate into the rest of the application. Therefore, the bootstrapping procedure from the next HEAAN iteration was used to speed up computation [BootstrappingHEAAN]. Bootstrapping takes advantage of the \textit{approximate computation} characteristic of HEAAN to evaluate the decryption formula approximately so an encryption of the original message can be obtained in a large ciphertext modulus. Hence, an approximation of the \textit{modular reduction} formula is implemented such that it can be efficiently evaluated using standard operations – where the error induced by the approximation is small enough to maintain precision.

Using the observation that the modular reduction function, $F(t) = [t]\_q$, is the identity near zero and periodic in $q$, bootstrapping uses a trigonometric function to approximate the function when $t = \langle \texttt{Ciphertext}, \texttt{SecretKey} \rangle$ is close to a multiple of $q$ (the ciphertext modulus). Specifically, using the \textit{sine} function given by Equation \ref{eq:bootstrapping}.

EQUATION

when $F(\langle \texttt{Ciphertext}, \texttt{SecretKey} \rangle) < \epsilon \cdot q$.

The Taylor polynomial can be used to approximate the trigonometric function so that it can be evaluated in the HE domain. The input, $t$, is bounded by $K \cdot q$ for some constant $K = O(\lambda)$, where $\lambda$ is the security parameter. The degree of the Taylor polynomial should be at least $O(K \cdot q)$ in order to make the error term small enough on the interval $(-K \cdot q, K \cdot q)$. Cheon et al.\ proposed using the \textit{Paterson-Stockmeyer} method to reduce the complexity of these calculations [CHEON, PATERSON]. However, the complexity of recryption grows exponentially with the depth of the decryption circuit, which still has a substantial impact.

Figure \ref{fig:bootstrapping} provides a graphical representation of the bootstrapping approximation.

## Optimisations

The principal optimisations of MeKKS came through data representation. When integrating SEAL, a Python wrapper for a C++ implementation was used. Consequently, the application was forced to handle large C++ objects, abstracted through the wrapper. Therefore, Python struggled to manipulate these objects efficiently, specifically when serialising data for transmission. However, since MeKKS was written in Python, the underlying functionality provided much more efficient manipulation as it could directly interact with the objects and their attributes.

Another optimisation came through the specialisation of the library. The project schedule ensured MeKKS was only implemented once the application’s core had been complete. As a result, the functionality required had been finalised and only needed operations were implemented for MeKKS. This further reduced the size of objects and removed any unnecessary computations, making execution more efficient. For example, the most expensive operations offered by SEAL are rotations. However, since the application did not use them, they were ignored during the implementation of MeKKS.