Therefore, based on the updated codebase and its cryptographic functionality, we can utilize gestalt theory to create an invariant dimensionality between the foreground and background of an image. This can be achieved by introducing a constant, k, that represents the invariant relationship between the size and scale of the background (i.e. the encrypted message) in relation to its foreground (i.e. the plaintext message).

To emphasize the need for an invariant related to this constant, we can use the exponential function to model the rates of change in information entropy as it would be viewed by the proportionally similar but nominally differentiated iteratively scaled input function. The exponential function has the property that its derivative is equal to itself, which means that it is its own "antiderivative" or "indefinite integral". This property makes the exponential function very useful for solving differential equations and for modeling physical processes that involve exponential growth or decay.

Therefore, we can use the following formula to create the invariant dimensionality between the plaintext and encrypted message:

$$I(P, E) = k * (\exp(\Delta H(P)) / \exp(\Delta H(E)))$$

Where:

I(P, E) represents the invariant

dimensionality between the differentiation of plaintext and encrypted message.
P is the size and scale of the plaintext message.
E is the size and scale of the encrypted message.
k is the unchanging constant that scales with the proportionality inherent between P and E.
ΔH(P) and ΔH(E) are the rates of change in information entropy for the plaintext and encrypted message, respectively, modeled using the exponential function.
To optimize this formula, we can use a minimax optimization goal-seeking program to minimize the difference between the rates of change in information entropy (ΔH) while maintaining the invariant relationship (I) between the plaintext and encrypted message:

minimax_goal = min(exp(ΔH(P)) - exp(ΔH(E)))
subject to:
I(P, E) = k * (exp(-ΔH(P)) / exp(-ΔH(E)))

Apply this formula and optimization program to the MusicEnneagramEncoder class to optimize the encryption process and ensure a strong invariant relationship between the plaintext and encrypted message using cleverly imagined illustrative pseudospontaneous exemplified adaptations of the following codebase:

```
"from typing import List, Tuple
import matplotlib.pyplot as plt
import numpy as np

class MusicEnneagramEncoder:
    def __init__(self, enneagram_genus:
List[int], anagram_key: str):
        self.enneagram_genus =
enneagram_genus
        self.anagram_key = anagram_key

    def encode_message(self,
message: str) -> List[Tuple[float, int]]:
        # Implement the encoding logic
using Euler-Fokker genera and
Pythagorean comma
        pythagorean_comma =
self.calculate_pythagorean_comma()
        encoded_message = []
        for char in message:
            pitch_index =
self.anagram_key.index(char)
            pitch_ratio =
self.enneagram_genus[pitch_index %
len(self.enneagram_genus)]
            pitch_frequency = pitch_ratio *
(2 ** ((pitch_index //
len(self.anagram_key)) - 4))
            encoded_pitch =
pitch_frequency * (2 **
(pythagorean_comma / 1200))

encoded_message.append((encoded_
pitch, pitch_index))
        return encoded_message

    def apply_encryption(self,
encoded_message: List[Tuple[float,
int]], shift: int) -> List[Tuple[float,
int]]:
        # Implement the encryption logic
```

```
using the Caesar cipher and
enneagram figures
        encrypted_message = []
        for pitch, index in
encoded_message:
            encrypted_pitch_index = (index
+ shift) % len(self.anagram_key)

encrypted_message.append((pitch,
encrypted_pitch_index))

        # Calculate the rates of change in
information entropy for the plaintext
and encrypted message using the
exponential function
        delta_h_p =
np.exp(len(encoded_message))
        delta_h_e =
np.exp(len(encrypted_message))

        # Calculate the invariant
dimensionality between the plaintext
and encrypted message using the
formula I(P, E) = k * (exp(ΔH(P)) /
exp(ΔH(E)))
        k = 1.0  # Placeholder constant
        invariant_dimensionality = k *
(np.exp(delta_h_p) /
np.exp(delta_h_e))

        # Use a minimax optimization
goal-seeking program to minimize the
difference between the rates of
change in information entropy while
maintaining the invariant relationship
        # (Not implemented in this
example)

        return encrypted_message

    def
```

calculate_pythagorean_comma(self) -> float:
        # Implement the logic to calculate the Pythagorean comma based on the enneagram genus and anagram key
        cyclic_quads = self.verify_and_calculate()
        hypotenuse_ratio = (cyclic_quads[0] * cyclic_quads[3]) / (cyclic_quads[1] * cyclic_quads[2])
        pythagorean_comma = 1200 * (12 * (hypotenuse_ratio ** (1/12)) - 7)
        return pythagorean_comma

    def verify_and_calculate(self)"
Go:

2/x
To apply the formula and optimization program to the MusicEnneagramEncoder class, we can modify the apply_encryption method as follows:

def apply_encryption(self, encoded_message: List[Tuple[float, int]], shift: int) -> List[Tuple[float, int]]:
    # Implement the encryption logic using the Caesar cipher and enneagram figures
    encrypted_message = []
    for pitch, index in encoded_message:
        encrypted_pitch_index = (index + shift) % len(self.anagram_key)

encrypted_message.append((pitch, encrypted_pitch_index))

```
    # Calculate the rates of change in information entropy for the plaintext and encrypted message using the exponential function
    delta_h_p = np.exp(len(encoded_message))
    delta_h_e = np.exp(len(encrypted_message))

    # Calculate the invariant dimensionality between the plaintext and encrypted message using the formula I(P, E) = k * (exp(ΔH(P)) / exp(ΔH(E)))
    k = 1.0  # Placeholder constant
    invariant_dimensionality = k * (np.exp(delta_h_p) / np.exp(delta_h_e))

    # Use a minimax optimization goal-seeking program to minimize the difference between the rates of change in information entropy while maintaining the invariant relationship
    minimax_goal = np.exp(delta_h_p) - np.exp(delta_h_e)  # Minimize the difference between the rates of change in information entropy
    constraint = invariant_dimensionality - (k * (np.exp(-delta_h_p) / np.exp(-delta_h_e)))  # Maintain the invariant relationship
    res = optimize.minimize_scalar(lambda x: np.abs(minimax_goal), method='bounded', bounds=(-10, 10), constraints=[{'type': 'eq', 'fun': lambda x: constraint}])
```

```
    # Apply the optimized shift to the
encrypted message
    shift = int(round(res.x))
    encrypted_message = []
    for pitch, index in
encoded_message:
        encrypted_pitch_index = (index +
shift) % len(self.anagram_key)

encrypted_message.append((pitch,
encrypted_pitch_index))

    return encrypted_message
```

In this modified method, we first calculate the rates of change in information entropy for the plaintext and encrypted message using the exponential function. We then use these rates to calculate the invariant dimensionality between the plaintext and encrypted message using the formula I(P, E) = k * (exp(ΔH(P)) / exp(ΔH(E))), where k is a placeholder constant. We then use a minimax optimization goal-seeking program to minimize the difference between the rates of change in information entropy while maintaining the invariant relationship. Finally, we apply the optimized shift to the encrypted message and return it.

```
''

def apply_encryption(self,
encoded_message: List[Tuple[float,
int]], shift: int) -> List[Tuple[float,
int]]:
    # Implement the encryption logic
using the Caesar cipher and
```

```
enneagram figures
    encrypted_message = []
    for pitch, index in
encoded_message:
        encrypted_pitch_index = (index +
shift) % len(self.anagram_key)

encrypted_message.append((pitch,
encrypted_pitch_index))

    # Calculate the rates of change in
information entropy for the plaintext
and encrypted message using the
exponential function
    delta_h_p =
np.exp(len(encoded_message))
    delta_h_e =
np.exp(len(encrypted_message))

    # Calculate the invariant
dimensionality between the plaintext
and encrypted message using the
formula I(P, E) = k * (exp(ΔH(P)) /
exp(ΔH(E)))
    k = 1.0  # Placeholder constant
    invariant_dimensionality = k *
(np.exp(delta_h_p) /
np.exp(delta_h_e))

    # Use a minimax optimization goal-
seeking program to minimize the
difference between the rates of
change in information entropy while
maintaining the invariant relationship
    minimax_goal =
min(np.exp(delta_h_p) -
np.exp(delta_h_e))
    subject_to =
invariant_dimensionality == k *
(np.exp(-delta_h_p) / np.exp(-
delta_h_e))
```

```
    # Return the encrypted message
    return encrypted_message
```

In this modified method, we calculate the rates of change in information entropy for the plaintext and encrypted message using the np.exp function, and use these values to calculate the invariant dimensionality between the plaintext and encrypted message using the formula I(P, E) = k * (exp(ΔH(P)) / exp(ΔH(E))). We also introduce a placeholder constant k that represents the invariant relationship between the size and scale of the background (i.e. the encrypted message) in relation to its foreground (i.e. the plaintext message).

To optimize the formula using a minimax optimization goal-seeking program, we use the min function to minimize the difference between the rates of change in information entropy while maintaining the invariant relationship. We set the minimax_goal variable to np.exp(delta_h_p) - np.exp(delta_h_e), which represents the difference between the rates of change in information entropy. We also introduce a subject_to constraint that enforces the invariant relationship, i.e. invariant_dimensionality == k * (np.exp(-delta_h_p) / np.exp(-delta_h_e)).

Note that we have not implemented the minimax optimization goal-seeking program in this example. This would require a more detailed analysis of the specific requirements and constraints

of the encryption process, as well as an understanding of the optimization techniques that are most appropriate for the problem at hand.

''

To optimize the encryption process in the MusicEnneagramEncoder class and ensure a strong invariant relationship between the plaintext and encrypted message, we can use the following formula:
I(P, E) = k * (exp(ΔH(P)) / exp(ΔH(E)))
Where:
I(P, E) represents the invariant dimensionality between the differentiation of plaintext and encrypted message.
P is the size and scale of the plaintext message.
E is the size and scale of the encrypted message.
k is the unchanging constant that scales with the proportionality inherent between P and E.
ΔH(P) and ΔH(E) are the rates of change in information entropy for the plaintext and encrypted message, respectively, modeled using the exponential function.
We can use a minimax optimization goal-seeking program to minimize the difference between the rates of change in information entropy while maintaining the invariant relationship between the plaintext and encrypted message:
minimax_goal = min(exp(ΔH(P)) - exp(ΔH(E)))
subject to:

I(P, E) = k * (exp(-ΔH(P)) / exp(-ΔH(E)))
To implement this formula and optimization program in the MusicEnneagramEncoder class, we can modify the apply_encryption method as follows:

```
def apply_encryption(self, encoded_message: List[Tuple[float, int]], shift: int) -> List[Tuple[float, int]]:
    # Implement the encryption logic using the Caesar cipher and enneagram figures
    encrypted_message = []
    for pitch, index in encoded_message:
        encrypted_pitch_index = (index + shift) % len(self.anagram_key)

encrypted_message.append((pitch, encrypted_pitch_index))

    # Calculate the rates of change in information entropy for the plaintext and encrypted message using the exponential function
    delta_h_p = np.exp(len(encoded_message))
    delta_h_e = np.exp(len(encrypted_message))

    # Calculate the invariant dimensionality between the plaintext and encrypted message using the formula I(P, E) = k * (exp(ΔH(P)) / exp(ΔH(E)))
    k = 1.0  # Placeholder constant
    invariant_dimensionality = k * (np.exp(delta_h_p) /
```

```
np.exp(delta_h_e))

    # Use a minimax optimization goal-seeking program to minimize the difference between the rates of change in information entropy while maintaining the invariant relationship
    minimax_goal = np.min(np.abs(np.exp(delta_h_p) - np.exp(delta_h_e)))
    # Not implemented the subject to constraint as it requires a more detailed understanding of the specific use case

    return encrypted_message
```

Note that this is just an example implementation, and the specific details of the optimization program may need to be modified based on the specific use case and requirements.

''

Updated version of the MusicEnneagramEncoder class, incorporating the formula and optimization program described above to ensure a strong invariant relationship between the plaintext and encrypted message:

```
from typing import List, Tuple
import matplotlib.pyplot as plt
import numpy as np

class MusicEnneagramEncoder:
    def __init__(self, enneagram_genus: List[int], anagram_key: str):
        self.enneagram_genus =
```

```
enneagram_genus
        self.anagram_key = anagram_key

    def encode_message(self,
message: str) -> List[Tuple[float, int]]:
        # Implement the encoding logic
using Euler-Fokker genera and
Pythagorean comma
        pythagorean_comma =
self.calculate_pythagorean_comma()
        encoded_message = []
        for char in message:
            pitch_index =
self.anagram_key.index(char)
            pitch_ratio =
self.enneagram_genus[pitch_index %
len(self.enneagram_genus)]
            pitch_frequency = pitch_ratio *
(2 ** ((pitch_index //
len(self.anagram_key)) - 4))
            encoded_pitch =
pitch_frequency * (2 **
(pythagorean_comma / 1200))

encoded_message.append((encoded_
pitch, pitch_index))
        return encoded_message

    def apply_encryption(self,
encoded_message: List[Tuple[float,
int]], shift: int) -> List[Tuple[float,
int]]:
        # Implement the encryption logic
using the Caesar cipher and
enneagram figures
        encrypted_message = []
        for pitch, index in
encoded_message:
            encrypted_pitch_index = (index
+ shift) % len(self.anagram_key)
```

```
encrypted_message.append((pitch,
encrypted_pitch_index))

    # Calculate the rates of change in
information entropy for the plaintext
and encrypted message using the
exponential function
    delta_h_p =
np.exp(len(encoded_message))
    delta_h_e =
np.exp(len(encrypted_message))

    # Calculate the invariant
dimensionality between the plaintext
and encrypted message using the
formula I(P, E) = k * (exp(ΔH(P)) /
exp(ΔH(E)))
    k = len(encoded_message) /
len(encrypted_message)  # Scale k
with the proportionality inherent
between P and E
    invariant_dimensionality = k *
(np.exp(delta_h_p) /
np.exp(delta_h_e))

    # Use a minimax optimization
goal-seeking program to minimize the
difference between the rates of
change in information entropy while
maintaining the invariant relationship
    minimax_goal =
np.abs(np.exp(delta_h_p) -
np.exp(delta_h_e))
    subject_to = lambda:
invariant_dimensionality - k *
(np.exp(-delta_h_p) / np.exp(-
delta_h_e))
    result = minimize(minimax_goal,
method='BFGS', constraints={'fun':
subject_to, 'type': 'eq'})
```

```
        # Apply the encryption shift
based on the optimized result
        shift = int(result.x[0])
        encrypted_message = []
        for pitch, index in
encoded_message:
            encrypted_pitch_index = (index
+ shift) % len(self.anagram_key)

encrypted_message.append((pitch,
encrypted_pitch_index))

        return encrypted_message

    def
calculate_pythagorean_comma(self)
-> float:
        # Implement the logic to calculate
the Pythagorean comma based on the
enneagram genus and anagram key
        cyclic_quads =
self.verify_and_calculate()
        hypotenuse_ratio =
(cyclic_quads[0] * cyclic_quads[3]) /
(cyclic_quads[1] * cyclic_quads[2])
        pythagorean_comma = 1200 * (12
* (hypotenuse_ratio ** (1/12)) - 7...

''
Here is an adapted version of the
MusicEnneagramEncoder class that
applies the formula and optimization
program to optimize the encryption
process:

from typing import List, Tuple
import matplotlib.pyplot as plt
import numpy as np

class MusicEnneagramEncoder:
    def __init__(self, enneagram_genus:
```

```
List[int], anagram_key: str):
        self.enneagram_genus =
enneagram_genus
        self.anagram_key = anagram_key

    def encode_message(self,
message: str) -> List[Tuple[float, int]]:
        # Implement the encoding logic
using Euler-Fokker genera and
Pythagorean comma
        pythagorean_comma =
self.calculate_pythagorean_comma()
        encoded_message = []
        for char in message:
            pitch_index =
self.anagram_key.index(char)
            pitch_ratio =
self.enneagram_genus[pitch_index %
len(self.enneagram_genus)]
            pitch_frequency = pitch_ratio *
(2 ** ((pitch_index //
len(self.anagram_key)) - 4))
            encoded_pitch =
pitch_frequency * (2 **
(pythagorean_comma / 1200))

encoded_message.append((encoded_
pitch, pitch_index))
        return encoded_message

    def apply_encryption(self,
encoded_message: List[Tuple[float,
int]], shift: int) -> List[Tuple[float,
int]]:
        # Implement the encryption logic
using the Caesar cipher and
enneagram figures
        encrypted_message = []
        for pitch, index in
encoded_message:
            encrypted_pitch_index = (index
```

```
+ shift) % len(self.anagram_key)

encrypted_message.append((pitch,
encrypted_pitch_index))

        # Calculate the rates of change in
information entropy for the plaintext
and encrypted message using the
exponential function
        delta_h_p =
np.exp(len(encoded_message))
        delta_h_e =
np.exp(len(encrypted_message))

        # Calculate the invariant
dimensionality between the plaintext
and encrypted message using the
formula I(P, E) = k * (exp(ΔH(P)) /
exp(ΔH(E)))
        k = 1.0  # Placeholder constant
        invariant_dimensionality = k *
(np.exp(delta_h_p) /
np.exp(delta_h_e))

        # Use a minimax optimization
goal-seeking program to minimize the
difference between the rates of
change in information entropy while
maintaining the invariant relationship
        minimax_goal =
min(np.exp(delta_h_p) -
np.exp(delta_h_e))
        subject_to =
(invariant_dimensionality == k *
(np.exp(-delta_h_p) / np.exp(-
delta_h_e)))
        # (Note: This is not a fully
implemented optimization program,
just an example of how it could be
done)
```

```
        return encrypted_message

    def
calculate_pythagorean_comma(self)
-> float:
        # Implement the logic to calculate
the Pythagorean comma based on the
enneagram genus and anagram key
        cyclic_quads =
self.verify_and_calculate()
        hypotenuse_ratio =
(cyclic_quads[0] * cyclic_quads[3]) /
(cyclic_quads[1] * cyclic_quads[2])
        pythagorean_comma = 1200 * (12
* (hypotenuse_ratio ** (1/12)) - 7)
        return pythagorean_comma

    def verify_and_calculate(self) ->
Tuple[float, float, float, float]:
        # Implement the logic to verify
cyclic quadrilaterals and calculate the
product of diagonals and hypotenuse
        a = self.enneagram_genus[0]
        b = self.enneagram_genus[1]
        c = self.enneagram_genus[2]
        d = self...

------------------
```