Mortgage-Backed Security Modelling

Jeff Nguyen

4/04/2021

University of Southern California
Marshall School of Business
GSBA 593 Independent Research
Spring 2021
Directed by Professor Mick Swartz

Abstract

The mortgage-backed securities market is one of the important credit markets that serve two purposes: 1) providing credit-granting institutions such as banks with a vehicle to limit their risks exposure and 2) providing investors with opportunities to generate returns by taking on those risks. One of the lingering effects of The Great Recession of 2008 is the eroded trust in credit agencies' abilities to accurately price mortgage-backed securities.

The proposed research aims to develop a quantitative methodology to model a mortgage-backed security. The focus of the study is to structure a mortgage-backed security (MBS) from a pool of mortgage loans and return a fair valuation of the structured deal while accounting for credit default risks.

The study will utilize credit data from sources such as the Federal Reserve Economic Data (FRED) and Fannie Mae. Credit data will be used to structure a Mortgage-Backed Security. The study will then simulate different credit defaults via Monte Carlo simulation methodology. The computational algorithms will be implemented in Python. The expected result is that the study will arrive at the fair value of the structured deal comparable to the actual results generated by the agency.

Introduction

https://www.sifma.org/resources/research/fixed-income-chart/

The US asset-backed securities market is a 11.2 trillion market as of 2020 (SIFMA). It covers an important consumer credit market of homeowners' loans. A well-functioned mortgage-backed securities market is instrumental in providing liquidity to important credit functions that drives the economy. The subprime crisis of 2008 has eroded trusts of investors in credit agencies' abilities to accurately priced mortgage-backed securities. Thus, the study seeks to showcase a quantitative and thus, objective, method to price these structured securities.

The proposed research consists of two integrated parts:

Part 1

Implementation of the structured security: The research will utilize sample credit data of an actual Mortgage-Backed Security from a market data source such as Fannie MAe. The objective is to then, design the structured security based on the cash flow of each period from the credit pool. The desired outcome is a presentation of the cash flows of each tranche in the structured deal and their corresponded components: principal and interest payments.

Part 2

Valuation of the structured security: The research will create a Monte Carlo simulation to simulate different credit default and prepayment scenarios to help determine the fair value of the structured deal. The desired outcome is metrics—yields and ratings—for each tranche in the structured deal.

The results will then be compared with the actual MBS's metrics to understand the effectiveness of the quantitative method and discuss other nuances of the differences between the quantitative method and reality.

Background

From the credit granting institutions' perspective, each structured security has two main components: the pool of assets—the underlying receivables from homeowners' loans—and the pool of liabilities—the structured securities.

Hypothetical Case Study

When Bank of America (BoA) grants credits to consumers, it guarantees the consumer's purchase of goods and services by making the immediate payments on behalf of the consumers to the vendors and receive periodic repayment from the consumers. BoA gets reimbursement for facilitating these transactions by charging a fee to the consumers as well as collecting interest rates on the credit amount. During the period from which the initial credit grant happens—when the consumers make the purchase—to the repayment of credit—when the consumer make the credit payment—BoA has exposed itself to two types of risks:

1. Default risk:

The risk that the consumer would not pay back the credit. Defaulted losses range from a percentage of to the full amount of the credit. This is an example of bad debt expense for BoA.

2. Prepayment risk:

The risk that consumer would pay back the credit earlier than expected. This scenario would result in a loss of expected income from interest payments for BoA. To limit its expose to these risks, BoA would package these credits together into a Special Purpose Vehicle. This vehicle is the structured security. The structured security enables a third party, the investors, to assume the majority of the risks—Dodd-Frank regulations require credit lender to assume some of the risks compared to 0% pre-2008—by making payments to the BoA to guarantees these credits and hoping to recoup the principal amount at the end of the credit term as well as earning a premium on periodic interest amounts (Practical Law). This fair premium is what the study seeks to arrive at.

There are different types of structured securities, such as standard and strips. The scope of this study only covers a two-tranche special purpose vehicle, where each tranche receives both the principal and interest payments from the Special Purpose Vehicle.

Design and Methodology

This study's structure is as follow:

1. Implementation of the Structured Securities:

a. Aggregation of the pool of outstanding credits

First, we determine the cash flows collections—from consumer's repayments—for each period based on each credits' notional, rate and term. In practice, a credit pool only comprises of a single class of loan–mortgage loans in our study–but theoretically, it can contains a mixture.

Second, we establish a structure of types and number of securities to be offered to potential investors. Each of these security is defined as a tranche. A structured deal, then, consists of a number and combination of tranches. In practice, there are different types of tranches. For example:

Classification by Payment Type:

- Standard Tranches: The Standard Tranche receive both interest and principal payments from a pool of loans.

- Interest-Only (IO) Strips: Interest-Only Tranches receive solely interest payments from the pool of loans. These type of tranches carry high amount of risk, mainly prepayment risks as once prepayment happens, these tranches receive no further interest payments.

- Principal-Only (PO) Strips: Opposite to IO Strips, Principal-Only Tranches receive solely principal payments from the credit pool. These types of tranches carry high amount of default risk, as they do not receive any interest payment to offset it.

The scope of this study will be contained in modelling standard tranches.

Classification by Risks: In addition to classification by payment types, tranches can be classified by type of risks. For example, a structured deal may contains multiple tranches: Class A, Class B, Class C, and so on. The Class B Tranche would be a subordinate of the Class A Tranche. The Class C TRanche would be a subordinate of the Class C one and so on. Subordination denotes the priority of each tranches in terms of receiving principal payments. In this case, Tranche B receives its principal payment after Tranche A. Consequently, Tranche B is exposed to higher default risk. AS a result, investors in Tranche B would expect a higher discount rate to compensate for additional risks.

b. Implement Waterfall cash flow model

Next, payment from the credit pool for each period is calculated and allocated to each tranche in the structure. The principal payment amount is allocated first. For example, if the total loan principal is $100,000,000$, Class A Tranche hypothetically could be allocated 70% or $70,000,000$, class B 20% $(20,000,000)$ and class C 10% $(10,000,000)$.

These payments are tracked via our Waterfall cash flow model and each payment is distributed to each tranches based on the structured deal agreement. Interest payments are allocated first and the remaining cash flow is allocated to pay down the principal balance.

Default Model For each payment period, a default model–using Monte Carlo Simulation–is incorporated into the Waterfall cash flow model. The default model will pseudorandomly alter cash flow: if there is default, cash payout from credit pool to the tranche will halt. For the purpose of this study, the default model will follow a lognormal distribution with pseudo-randomly generated default parameters.

2. Valuation of the Structured Securities:

In practice, investors utilize key metrics such as credit rating, structured deal's discount rates and Weighted Average Life (WAL) to make investment decisions on Mortgage-Back Securities. The credit rating is external to the structured deal and is determined by a third party such as the Big 3 major credit agencies–Standard & Poor's Global Ratings, Moody's, and Fitch Ratings. The discount rate and WAL metrics are native to the structured deal. The purpose of these metrics is to provide fair value assessment to the structured deal to protect the issuing institution as well as the investors.

In this case study, we will run two-tiered Monte Carlo Simulation to simulate the random component of the default risk model. The goal is to arrive at fair valuation of these metrics based on the Law of Large Numbers, i.e. thousands of default scenarios can potentially give a wholesome picture of the true valuation of the security.

3. Comparison to the actual agency's metrics.

Once we arrive at fair valuation using the quantitative model for the structured security, we will compare the result to the actual valuation using the source, e.g. Fannie Mae and discuss the differences.

Implementation of the Quantitative Model

Waterfall Cash Flow Model

We recall that the Waterfall Cash Flow Model keeps track of all cash flow from the credit pool for each period. Below shows the step by step of the quantitative implementation of the Waterfall Cash Flow Model.

Base Tranche Implementation

First, in Python, we create an abstract base class called **Tranche** and initialize it with notional–representing the total notional value of the structured deal–and rate–representing the discount rate of the deal. This base class will also have a subordination flag to identify the priority of this tranche. Full implementation of this Tranche can be viewed in Exhibit A1 in Appendix A.

Standard Tranche Implementation

Next, we create a derived class named **StandardTranche** representing a Standard Tranche type for the purpose of this study. As recall, in practice, there can be other type of tranches such as "PO Strips" and "IO Strips".

The StandardTranche is initialized with a notional value and is used to keep track of all credit payment made to the structured deal including principal payments and interest payments for each period. The following methods is created within the StandardTranche class to execute the above operations. Full implementation of this class is presented in Exhibit A2.

increaseTimePeriod():

This method increases the current time period by 1 to cycle through all payment period once tranches' operations have been completed.

makePrincipalPayment():

This method keep track of principal payment for each corresponding period. This method is called once for each period. When the notional balance of the tranche reaches 0, i.e. the tranche has been fully paid, this method will stop accepting credit payments and send an notification to the Waterfall Model engine.

makeInterestPayment():

Similar to makePrincipalPayment() method, this method record interest payment for the current period. One major difference is that when interest inflow from the credit pool is less than the interest due amount, i.e. there is not enough cash inflow to pay the full balance, the method will register an interest shortfall amount.

notionalBalance():

This method calculates the remaining notional balance of the tranche for the current period based on the original notional balance, the cumulative principal payments to date and any interest shortfalls.

interestDue():

This method calculates the interest due for the current payment period.

Structured Deal Implementation:

The Structured Deal implementation involves operating on a composition of Tranche classes objects with the following methods:

addTranche():

This method add tranches to the structured deal.

makePayments():

This method is used to allocate credit payments to tranches. It does so by looping through all tranches and execute the following operations:

- It cycles through all interest payments first, attempting to pay off each tranche from available payment from credit pool. If there exists a short fall, this amount is recorded and added onto the interest due for the next period. For the purpose of this study, we naively assume there is no compounding interest on the shortfall.

- If there is leftover credit pool payment after making all interest payments for all tranches, this method would attempt to pay the principal amount for each tranche. If the tranche type is **Sequential**, the method would pay each tranche principal in full before moving to the next tranche. If the tranche type is **Pro Rata**, the method would allocate a predetermined percentage of available cash to each tranche based on the structured deal agreement.
- Once all interest and principal payments have been made successfully, the method will record any leftover cash amount in a reserve account to be cumulatively added to the next period payment from the credit pool. Again, this study will naively assume that this cash amount will earn no interest when in practice, it will earn a compounding rate.

getWaterfall():

For each period, this method will return the ledger of each tranche including but not limited to the amount of interest due, interest paid, interest shortfall, principal paid and remaining balance.

Waterfall Cash Flow Model Execution

This method execute the Waterfall Model by performing the following operations. A full implementation of this method is presented in Exhibit A3.

- Starting from the beginning period, i.e. time $t = 0$: For each period, interact with the credit pool to receive the aggregated payments.
- Cycle through each tranche in the Structured Deal and allocate the appropriate payment amount.
- Record payments and balances with the credit pool as well as the Structured Deal.

Waterfall Model Metrics

Metrics for Waterfall Model

Simulation for Structured Deal Valuation

Appendix

Appendix A

Exhibit A1

```
# Tranche Base class
class Tranche(object):
    def __init__(self, notional, rate, subordinationFlag):
        self._notional = notional
        self._rate = rate
        self._subordinationFlag = subordinationFlag
        self._r = 0  # Record IRR
        self._dirr = 0  # Record DIRR
```

```
        self._dirrLetter = None  # Record DIRR in letter form
        self._al = 0  # Record Average Life

    ############################################################
    # Decorators to define and set values for instance variables

    # Decorator to create a property function to define the attribute notional
    @property
    def notional(self):
        return self._notional

    # Decorator to set notional value
    @notional.setter
    def notional(self, inotional):
        self._notional = inotional  # Set instance variable notional from input

    # Decorator to create a property function to define the attribute rate
    @property
    def rate(self):
        return self._rate

    # Decorator to set rate value
    @rate.setter
    def rate(self, irate):
        self._rate = irate  # Set instance variable rate from input

    # Decorator to create a property function to define the attribute
    # subordinationFlag
    @property
    def subordinationFlag(self):
        return self._subordinationFlag

    # Decorator to set subordinationFlag value
    @subordinationFlag.setter
    def subordinationFlag(self, isubordinationFlag):
        # Set instance variable subordinationFlag from input
        self._subordinationFlag = isubordinationFlag

    # Decorator to create a property function to define the attribute r (IRR)
    @property
    def r(self):
        return self._r

    # Decorator to set r (IRR) value
    @r.setter
    def r(self, ir):
        self._r = ir  # Set instance variable r (IRR) from input

    # Decorator to create a property function to define the attribute dirr
    @property
    def dirr(self):
        return self._dirr
```

```
    # Decorator to set dirr value
    @dirr.setter
    def dirr(self, idirr):
        self._dirr = idirr  # Set instance variable dirr from input

    # Decorator to create a property function to define the attribute dirrLetter
    @property
    def dirrLetter(self):
        return self._dirrLetter

    # Decorator to set dirrLetter value
    @dirrLetter.setter
    def dirrLetter(self, idirrLetter):
        # Set instance variable dirrLetter from input
        self._dirrLetter = idirrLetter

    # Decorator to create a property function to define the attribute al
    @property
    def al(self):
        return self._al

    # Decorator to set al value
    @al.setter
    def al(self, ial):
        self._al = ial  # Set instance variable al from input
    ############################################################
    # Add instance methods

    # Return total payment made for each period t
    def paymentPerPeriod(self, t):
        raise NotImplementedError('Must override StandardTranche')

    # Return total payment period:
    def totalPaymentPeriod(self):
        raise NotImplementedError('Must override StandardTranche')

    # Calculate Internal Rate of Return (IRR)
    # This is the interest rate that results in the present value of all
    #   (monthly) cash flows being equal to the initial investment amount.
    #   The equation for this is as follows (where C is
    #   the payment at time t, T is the total number of periods, and r is the
    #   monthly rate):
    #       0 = -C_0 + \sum_{t=1}^{T} \frac{C_t}{(1+r)^t}
    def IRR(self):
        # Override method from StandardTranche
        period = self.totalPaymentPeriod()
        # Override method from StandardTranche to find CF per period
        cf = [self.paymentPerPeriod(t) for t in range(1, period)]
        # insert original notional value as cash outflow
        cf.insert(0, (- self.notional))
        self.r = numpy_financial.irr(cf) * 12  # Return annualized IRR
        return self.r
```

```
    # Calculate Reduction in Yield (DIRR)
    # This is the tranche rate less the annual IRR. Essentially,
    # the annual tranche rate is the annualized rate of return that the
    # investor expects to earn whereas the IRR is the realized return.
    # DIRR specifies how much the investor lost out on (hence, its maximum is
    # 100% + the tranche rate).
    # Additionally, DIRR is used to give a letter rating to the security.
    def DIRR(self):
        self.dirr = round(self.rate - self.IRR(), 6)
        return self.dirr

    # Get DIRR in letter form
    def DIRRLetter(self):
        self.dirrLetter = self.getRating(self.dirr)
        return self.dirrLetter

    # Calculate Average Life (AL)
    # The AL of the security is the average time that each dollar of its
    # unpaid principal remains unpaid.
    # This is the inner product of the time period numbers (0, 1, 2, 3, etc.)
    # and the principal payments, divided by the initial principal.
    # For example, if you have the principal payment list as follows: [10000,
    #   90000, 35000, 0], the AL would be
    # (0*0 + 1*10000 + 2*90000+2*35000+3*35000+4*0)/100000. If the loan was not
    # paid down (balance != 0), then AL is infinite - in this case, return None.
    def AL(self):
        raise NotImplementedError('Must override StandardTranche')

    ############################################################
    # Add class methods

    ############################################################
    # Add static methods

    # Look up rating based on DIRR
    # Method: sort the dict, then get the smallest value at which value >= DIIR.
    # Lookup key from the inverted dict.
    @staticmethod
    def getRating(dirr):
        dirr = dirr / 100
        ratings = {'Aaa': 0.06,
                   'Aa1': 0.67,
                   'Aa2': 1.3,
                   'Aa3': 2.7,
                   'A1': 5.2,
                   'A2': 8.9,
                   'A3': 13,
                   'Baa1': 19,
                   'Baa2': 27,
                   'Baa3': 46,
                   'Ba1': 72,
                   'Ba2': 106,
                   'Ba3': 143,
```

```
                    'B1': 183,
                    'B2': 231,
                    'B3': 311,
                    'Caa': 2500,
                    'Ca': 10000}
        sorted_key = dict(sorted(ratings.items(), key=lambda k: k[1],
        reverse=False))  # Sort dict
        # Smallest value at which v>= dict
        closest_value = min(sorted_key.values(), key=lambda v: v >= dirr)
        # Get the inverted dict to lookup key
        ratingsInv = {v: k for k, v in ratings.items()}
        return ratingsInv[closest_value] if not numpy.isnan(dirr) else 'Ca'
    ###########################################################
```

Exhibit A1

```
# Standard Tranche class
# Standard tranches receive both interest and principal payments
# from the pool of loans.
class StandardTranche(Tranche):
    def __init__(self, notional, rate, subordinationFlag):
        # Invoke base class init
        super(StandardTranche, self).__init__(notional, rate, subordinationFlag)
        self._principalPaid = {0: 0}  # Record principal payment
        self._principalShortFall = {0: 0}  # Record principal payment
        self._principalDue = {0: 0}  # Record principal due for each period
        self._interestPaid = {0: 0}  # Record interest payment
        self._interestShortFall = {0: 0}  # Record interest shortfall
        self._interestDue = {0: 0}  # Record interest due for each period
        # Record notional balance owed to the tranche
        self._notionalBalance = {0: notional}
        self._timePeriod = 1
        self._interestHasBeenPaid = False
        self._principalHasBeenPaid = False
        self._r = 0  # Record IRR
        self._dirr = 0  # Record DIRR
        self._dirrLetter = None  # Record DIRR in letter form
        self._al = 0  # Record Average Life
    ###########################################################
    # Decorators to define and set values for instance variables

    # Decorator to create a property function to define the attribute notional
    @property
    def notional(self):
        return self._notional

    # Decorator to set notional value
    @notional.setter
    def notional(self, inotional):
        self._notional = inotional  # Set instance variable notional from input

    # Decorator to create a property function to define the attribute rate
```

```
    @property
    def rate(self):
        return self._rate

    # Decorator to set rate value
    @rate.setter
    def rate(self, irate):
        self._rate = irate  # Set instance variable rate from input

    # Decorator to create a property function to define the attribute
    # subordinationFlag
    @property
    def subordinationFlag(self):
        return self._subordinationFlag

    # Decorator to set subordinationFlag value
    @subordinationFlag.setter
    def subordinationFlag(self, isubordinationFlag):
        # Set instance variable subordinationFlag from input
        self._subordinationFlag = isubordinationFlag

    # Decorator to create a property function to define the attribute
    # interestDue
    @property
    def interestDue(self):
        return self._interestDue

    # Decorator to set interestDue value
    @interestDue.setter
    def interestDue(self, iinterestDue, t):
        # Set instance variable interestDue from input
        self._interestDue[t] = iinterestDue

    # Decorator to create a property function to define the attribute
    # interestPaid
    @property
    def interestPaid(self):
        return self._interestPaid

    # Decorator to set interestPaid value
    @interestPaid.setter
    def interestPaid(self, iinterestPaid, t):
        # Set instance variable interestPaid from input
        self._interestPaid[t] = iinterestPaid

    # Decorator to create a property function to define the
    # attribute interestShortFall
    @property
    def interestShortFall(self):
        return self._interestShortFall

    # Decorator to set interestShortFall value
    @interestShortFall.setter
```

```
    def interestShortFall(self, iinterestShortFall, t):
        # Set instance variable interestShortFall from input
        self._interestShortFall[t] = iinterestShortFall

    # Decorator to create a property function to define the attribute
    # interestHasBeenPaid
    @property
    def interestHasBeenPaid(self):
        return self._interestHasBeenPaid

    # Decorator to set interestHasBeenPaid value
    @interestHasBeenPaid.setter
    def interestHasBeenPaid(self, iinterestHasBeenPaid):
        # Set instance variable interestHasBeenPaid from input
        self._interestHasBeenPaid = iinterestHasBeenPaid

    # Decorator to create a property function to define the
    # attribute principalDue
    @property
    def principalDue(self):
        return self._principalDue

    # Decorator to set principalDue value
    @principalDue.setter
    def principalDue(self, iprincipalDue, t):
        # Set instance variable principalDue from input
        self._principalDue[t] = iprincipalDue

    # Decorator to create a property function to define the
    # attribute principalPaid
    @property
    def principalPaid(self):
        return self._principalPaid

    # Decorator to set principalPaid value
    @principalPaid.setter
    def principalPaid(self, iprincipalPaid, t):
        # Set instance variable principalPaid from input
        self._principalPaid[t] = iprincipalPaid

    # Decorator to create a property function to define the attribute
    # principalShortFall
    @property
    def principalShortFall(self):
        return self._principalShortFall

    # Decorator to set principalShortFall value
    @principalShortFall.setter
    def principalShortFall(self, iprincipalShortFall, t):
        # Set instance variable principalShortFall from input
        self._principalShortFall[t] = iprincipalShortFall

    # Decorator to create a property function to define the attribute
```

```
    # principalHasBeenPaid
    @property
    def principalHasBeenPaid(self):
        return self._principalHasBeenPaid

    # Decorator to set principalHasBeenPaid value
    @principalHasBeenPaid.setter
    def principalHasBeenPaid(self, iprincipalHasBeenPaid):
        # Set instance variable principalHasBeenPaid from input
        self._principalHasBeenPaid = iprincipalHasBeenPaid

    # Decorator to create a property function to define the attribute
    # notionalBalance
    @property
    def notionalBalance(self):
        return self._notionalBalance

    # Decorator to set notionalBalance value
    @notionalBalance.setter
    def notionalBalance(self, inotionalBalance, t):
        # Set instance variable notionalBalance from input
        self._notionalBalance[t] = inotionalBalance

    # Decorator to create a property function to define the attribute timePeriod
    @property
    def timePeriod(self):
        return self._timePeriod

    # Decorator to set timePeriod value
    @timePeriod.setter
    def timePeriod(self, itimePeriod):
        # Set instance variable timePeriod from input
        self._timePeriod = itimePeriod
    # Decorator to create a property function to define the attribute r (IRR)
    @property
    def r(self):
        return self._r

    # Decorator to set r (IRR) value
    @r.setter
    def r(self, ir):
        self._r = ir  # Set instance variable r (IRR) from input

    # Decorator to create a property function to define the attribute dirr
    @property
    def dirr(self):
        return self._dirr

    # Decorator to set dirr value
    @dirr.setter
    def dirr(self, idirr):
        self._dirr = idirr  # Set instance variable dirr from input
```

```
    # Decorator to create a property function to define the attribute dirrLetter
    @property
    def dirrLetter(self):
        return self._dirrLetter

    # Decorator to set dirrLetter value
    @dirrLetter.setter
    def dirrLetter(self, idirrLetter):
        # Set instance variable dirrLetter from input
        self._dirrLetter = idirrLetter

    # Decorator to create a property function to define the attribute al
    @property
    def al(self):
        return self._al

    # Decorator to set al value
    @al.setter
    def al(self, ial):
        self._al = ial  # Set instance variable al from input
    ############################################################
    # Add instance methods

    # Increase the current time period of the object
    def increaseTimePeriod(self):
        self.timePeriod += 1
        self.interestHasBeenPaid = False
        self.principalHasBeenPaid = False
        return self.timePeriod

    # Record principal due, payment and shortfall for the current tranche
    # time period
    # Can only be called once, otherwise raised an error
    def makePrincipalPayment(self, t, prinDue, prinPaid, prinShortFall):
        if self._principalHasBeenPaid:
            raise Exception(f'Payment already made for this period.')
        else:
            # Record principal due for the period
            self.principalDue[t] = prinDue
             # Record principal paid for the period
            self.principalPaid[t] = prinPaid
            # Record principal short fall for the period
            self.principalShortFall[t] = prinShortFall
            self.principalHasBeenPaid = True

        if self.calc_notionalBalance(t) == 0:
            raise Exception('Zero balance. All paid.')

    # Record interest payment for the current tranche time period
    # Record interest payment for the current tranche time period
    # Can only be called once, otherwise raised an error
    # If the interest amount is less than the current interest due:
    #   In this case, the missing amount needs to be recorded separately as
```

```
    # an interest shortfall.
    def makeInterestPayment(self, t, paidAmount):
        if self._interestHasBeenPaid:
            raise Exception(f'Payment already made for this period.')
        else:
            # Record interest due for the period
            self.interestDue[t] = self.calc_interestDue(t)
            # Record interest paid for the period
            self.interestPaid[t] = paidAmount
            # Record interest short fall for the period
            self.interestShortFall[t] = self.interestDue[t] - paidAmount
            self.interestHasBeenPaid = True

        if self.interestDue[t] == 0:
            raise Exception('Zero Balance. All Paid.')

    # Return the amount of notional still owed to the tranche for the current
    # time period (after any payments made).
    # You can calculate this based on the original notional,
    # the sum of all the principal payments already made, and any interest
    # shortfalls.
    def calc_notionalBalance(self, t):
        if not t == 0:
            self.notionalBalance[t] = self.notional - sum([val for val in self.principalPaid.values()])
        return self.notionalBalance[t]

    # Return the amount of interest due for the current time period.
    def calc_interestDue(self, t):
        if not t == 0:
            self.interestDue[t] = self.calc_notionalBalance(t-1) * self.monthlyRate(self.rate) + \
                                  self.interestShortFall[t-1]
        return self.interestDue[t]

    # Return total payment made for each period t
    def paymentPerPeriod(self, t):
        return self.interestPaid[t] + self.principalPaid[t]

    # Return total payment period:
    def totalPaymentPeriod(self):
        return max(len(self.interestPaid), len(self.principalPaid))

    # Calculate Average Life (AL)
    # The AL of the security is the average time that each dollar of its unpaid
    # principal remains unpaid.
    # This is the inner product of the time period numbers (0, 1, 2, 3, etc.)
    # and the principal payments, divided by the initial principal.
    # For example, if you have the principal payment list as follows:
    # [10000, 90000, 35000, 0], the AL would be
    # (0*0 + 1*10000 + 2*90000+2*35000+3*35000+4*0)/100000.
    # If the loan was not paid down (balance != 0),
    # then AL is infinite - in this case, return None.
    def AL(self):
        al = sum([t * principalPayment / self.notional for t, principalPayment in self.principalPaid.it
```

```
        self.al = al if not math.isinf(al) else None
        return self.al

    # Reset the tranche to time t=0
    def reset(self):
        self._principalPaid = {0: 0}  # Record principal payment
        self._principalShortFall = {0: 0}  # Record principal payment
        self._principalDue = {0: 0}  # Record principal due for each period
        self._interestPaid = {0: 0}  # Record interest payment
        self._interestShortFall = {0: 0}  # Record interest shortfall
        self._interestDue = {0: 0}  # Record interest due for each period
        # Record notional balance owed to the tranche
        self._notionalBalance = {0: self.notional}
        self._timePeriod = 1
        self._interestHasBeenPaid = False
        self._principalHasBeenPaid = False

    ###################################

    ############################################################
    # Add class methods

    ############################################################
    # Add static methods

    # Calculate monthly rate based on annual rate
    @staticmethod
    def monthlyRate(annual_rate):
        return annual_rate / 12
    ############################################################
```

Exhibit A3

```
def doWaterfall(loans, tranches):
    tranches.reset()
    loans.reset()

    logging.debug(f'Doing work on {tranches.__repr__()}')

    ledger = [tranches.getWaterfall(0)]
    t = 0
    while loans.activeLoanCount(t) > 0:
        # Increase the time period on the StructuredSecurities object
        # (which will, in turn, increase for all the tranches).
        tranches.increaseTranchesTimePeriod()
        t += 1

        # Ask the LoanPool for its total payment for the current time period.
        # This is the paymentDue amount plus asset recovery value from defaults
        collections = loans.paymentDue(t)
        recoveries = loans.checkDefaults(t)
```

```
        # Ask the LoanPool for its total principal due for the current
        # time period.
        principalCollected = loans.principalDue(t)
        # Save the principal due
        tranches.save_principalCollected(t, principalCollected)
        # Pay the StructuredSecurities with the amount provided by the LoanPool.
        tranches.makePayments(collections + recoveries)
        # Call getWaterfall on both the LoanPool and StructuredSecurities
        # objects and save the info into two variables.
        ledger.append(tranches.getWaterfall(t))

    # For each tranches, save its metrics as a tuple of 4 values
    # Method is to call these tranche's methods. Calling these methods also
    # save the metric in the tranche's parameters.
    # For a 2 tranches securities, the metrics list will be a list of 2 tuples,
    # each tuples has 4 values
    # The metrics to be saved are: IRR, DIRR, letter DIRR rating, and AL
    metrics = [(tranche.IRR(), tranche.DIRR(), tranche.DIRRLetter(),
    tranche.AL()) for tranche in tranches.tranches]

    return ledger, metrics
```

Appendix B

References

Practical Law. (n.d.). Retrieved November 25, 2020, from https://ca.practicallaw.thomsonreuters.com/3-502-8950?transitionType=Default Rutledge, Ann, and Sylvain Raynes. Elements of Structured Finance. Oxford University Press. 2010. SIFMA. (n.d.). Retrieved November 25, 2020, from https://www.sifma.org/resources/research/fixed-income-chart/