9815 – Python Project

Case Study: Asset Backed Security Modeling

The final project case study is the concluding piece of this course. The topic of this case study is Asset Backed Security modeling. This will give you the opportunity to satisfyingly apply all of what you've learned about Python into a useful project in finance. The project will build upon the Loan classes and functionality you've created.
This project consists of three separate parts, each of which builds off the previous. The three parts are described below (with their respective project weights), and will each be explained in detail later. Note that you may choose to opt out of any of the below project parts, at the cost of the grade for that part.

1. Implement the actual asset-backed securities (the *liabilities*) in addition to the *Waterfall* mechanism that calculates the cashflows at each time period. The objective is to create well-designed tranche classes which will seamlessly work with your existing Loan classes. The outcome is to be able to take an input CSV of loan data and output a CSV with the all the cashflows at each time period (the *Waterfall*). This is 50% of the final project grade.

2. Implement metrics on the Waterfall. This includes Internal Rate of Return (IRR), Reduction in Yield (DIRR), and Average Life (AL). The objective and outcome is to be able to calculate and provide useful metrics on the structure. This is 15% of the final project grade.

3. The last part is to value and rate the ABS. This entails creating a Monte Carlo simulation to simulate thousands of different credit default scenarios, all of which help determine the rating of the structure. The objective here is to get a taste of implementing an actual Monte Carlo simulation for finance in Python, utilizing your existing classes, random number generation and multiprocessing. The outcome will be a rate, rating, and Weighted Average Life (WAL) for each tranche of our very simple structure. This is 35% of the final project grade.

The subject of Structured Finance is quite complex. The purpose of this project is not to teach you everything to do with Structured Finance (which is the topic of a full MFE course), but to apply Python concepts and techniques in a useful way, while giving you a taste of the concepts of Structure Finance. Consequently, this project will present an overly simplistic ABS model and will gloss over many of the more involved techniques (such as credit default and prepayment modeling); we will point out situations where the 'real-world' approach is more complex or may differ somewhat from this project's specification. Hopefully, you will come away from this project with a good sense of the power of Python and how it is used in banks, hedge funds, and other financial institutions.

Each part of this project will provide a rough set of steps to follow. Following these steps exactly is not a requirement, if you can come up with a better design, as long as your overall functionality is the same. Moreover, the provided steps are not comprehensive – you will need to make decisions as to which Python techniques to use. You should apply everything you have learned as and when appropriate (i.e., exception handling, logging, list comprehensions, dicts, using your Timer class or decorator to help optimize your code, etc.).

Background

Every structured deal consists of a pool of assets (the Loans) and a group of liabilities (the asset-backed securities). The objective of structuring is to create and sell customized securities to investors, which are backed by the pool of loans. The following example demonstrates both the motivation for this and the approach taken to achieve it:

Bank A has made a large number of loans. When Bank A makes these loans, it becomes exposed to two types of risk:
1. Default risk: The risk that the borrowers will fail to pay back the loan. This can occur if borrowers lose their jobs or some other life event that causes them to be unable to repay the loan. In the extreme case, this can be a result of an economic downturn that causes a large number of borrowers to be unable to repay.

2. Prepayment risk: The risk that the borrowers will pay back the loan early. This can result in a loss of expected interest income, especially if interest rates have decreased. Note that the most common reason for borrowers to prepay is if they refinance their loan due to lower interest rates. Another, less common reason to prepay, is if borrowers wish to get rid of the debt (i.e., they have a windfall and can afford the large cash payment).

Bank A does not wish to be exposed to these risks for the life of the loans; it just wants to make quick money when making the loan, with little to no risk. To mitigate this risk, it can sell the loans, as *structured securities*, to investors. Bank A then makes a quick profit on the *spread*.
Investors are willing to take on the risks of the loans as they hope to receive a decent return on their investment.
The process of creating the securities that are sold to the investors is called 'structuring'. The following are the (simplified) set of steps for structuring the securities:
1. Aggregate a pool of outstanding loans. Generally, a pool only contains a single class of loans (i.e., all car loans, all mortgages, or all student loans), but can theoretically contain a mixture.

2. Decide on the types and number of securities that will be offered to investors. Each security is called a *tranche*. A deal may consist of any number and combination of tranches. The entire structure of tranches is called the *deal*. There are many types of tranches. Here are the basic ones:

o Standard tranches receive both interest and principal payments from the pool of loans.

o Interest Only (IO) tranches (aka IO strips) only receive interest payments from the pool of loans. These tranches have much more prepayment risk than standard tranches (since a prepayment means no more interest).

o Principal Only (PO) tranches (aka PO strips) only receive principal payments from the pool of loans. These tranches are much more sensitive to default risk, as they never receive any interest payments to make up for it.

Our model will discuss and contain standard tranches only, for simplicity.

Tranches can also be classified by risk: For example, a deal may contain two (or more) standard tranches, Class A and Class B. The Class B tranche would be a *subordinate* of the Class A in that it receives its principal payments after Class A; this exposes Class B to higher default risk, since it always receives its principal payments after Class A (and only if there is enough cash left). Of course, this means that Class B investors would expect a better rate of return to compensate for the additional risk.

3. Allocate principal amounts to each tranche in the structure. The loan pool has a total (aggregated) principal. Each standard tranche in the structure will be allocated a portion of the total principal. For example, if the total loan principal is $10,000,000, Class A may be allocated 80% ($8,000,000) and Class B may be allocated 20% ($8,000,000).

4. Create a Waterfall cashflow model of the proposed structure. This tracks the cashflows out of the loan pool at each time period (i.e., monthly) and distributes them appropriately to each tranche in the structure. In general, interest payments to each tranche take priority, and principal payments are made with the remaining funds. More details on this later.

5. Decide on appropriate default and prepayment models that (as accurately as possible) reflect reality. Incorporate these models into the Waterfall. This will affect the cashflows, as a default will lower the monthly payments coming in from the loan pool (resulting in less cash to distribute to the tranches).

The default models will usually follow a distribution (i.e., a lognormal distribution) and have a random component to them.

6. Every ABS has a Rating (which are usually performed by a credit rating agency, such as Moody's or S&P). This Rating, the interest rate, and something called Weighted Average Life (WAL) are metrics used by investors to determine whether or not to invest in the securities.

The goal of these metrics is to provide a fair value to both sides of the deal: The seller (Bank A) wants the lowest possible cost and the investors which want the highest return possible. The fair value is one that takes all the risk components into account and neither shortchanges the seller nor rips off the investor. To generate these metrics, one can run a two-tiered Monte Carlo simulation to simulate the random component of the default risk models. Running thousands of scenarios will (hopefully) arrive at the fair value of the three metrics.

7. Once the deal is structured, investors can buy the newly structured securities. As mentioned above, each tranche (security) has a notional. An investor can buy an entire tranche or a piece of a tranche. For example, if the Class A standard tranche has a notional of $8,000,000, a single investor can buy the entire tranche for $8,000,000 or a piece of it (i.e., $500,000). Once Bank A sells all the securities, it has fully recouped its money (that it originally lent to the borrowers of the loan pool) and no longer has any risk exposure on these loans.

8. At every time period (i.e., monthly), the cashflows from the loan pool are distributed to each tranche, depending on the agreed-upon structure. The hope of each investor is to fully recoup its principal and yield an additional return by the end of the life of the security. Investors who like higher risk for a potential higher return are more likely to invest in a lower-class (riskier) tranche, while those who want a low-risk small return are more likely to invest in a Class A tranche.

Part 1: The Waterfall

As touched on in the Background section, the Waterfall tracks all the cashflows through each time period. Essentially, the pool of loans (the assets) will provide a monthly cashflow (resulting from the individual loan monthly payments) and that cashflow is allocated to each of the tranches in the structure (the liabilities).

We've already implemented the asset side of the Waterfall. To create the Waterfall, we first need to implement the liabilities (the tranches). The following are the rough guidelines for this implementation (you should implement using all the knowledge and best practices gained throughout the course):

1. Create an abstract base class called **Tranche**. It should be initialized with *notional* and *rate*. Additionally, it should have a *subordination* flag. This flag can either be letters of the alphabet or numbers.

2. Derive a class called **StandardTranche**. This will be the only derived-class for this project. However, in practice, you can derive other tranche types (i.e., for IO or PO).

- Standard tranches start off with a certain notional and need to be able to keep track of all payments made to it. More specifically, they need to be able to keep track of all interest payments and all principal payments made to it, and at what time period. To this end, implement the following methods:

- **increaseTimePeriod**: This should increase the current time period of the object by 1 (starts from 0).
- **makePrincipalPayment**: This should record a principal payment for the current object time period. This should only be allowed to be called once for a given time period (raise an error otherwise). Additionally, if the current notional balance is 0, the function should not accept the payment (raise an error).
- **makeInterestPayment**: This should record a principal payment for the current object time period. This should only be allowed to be called once for a given time period (raise an error otherwise). Additionally, if the current interest due is 0, the function should not accept the payment (raise an error). One special case that needs handling is what happens 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*.
- **notionalBalance**: This should return the amount of notional still owed to the tranche for the current time period (after any payments made). Hint: You can calculate this based on the original notional, the sum of all the principal payments already made, and any interest shortfalls.
- **interestDue:** This should return the amount of interest due for the current time period. Hint: This can be calculated using the notional balance of the previous time period and the rate.
- **reset**: This should reset the tranche to its original state (time 0).

3. Create a class called **StructuredSecurities**. This will be a composition of Tranche objects (similar to how **LoanPool** is a composition of **Loan**s). It should be initialized with a total Notional amount. ○ Create a 'factory method' to add tranches to a **StructuredSecurity** object. This can be called **addTranche**, and it will instantiate and add the tranche to the **StructuredSecurity** object's internal list of tranches. The method needs to know the tranche class (always **StandardTranche** in this project), the percent of notional, the rate, and the subordination level.
o Add a method that flags 'Sequential' or 'Pro Rata' modes on the object – more on this in a moment.
o Add a method that increases the current time period for each tranche.
○ Create a method called **makePayments**. This should have a *cash_amount* parameter. Cycle through all the tranches, in order of subordination. It should do so within the following guidelines: ✦ It should cycle through all interest payments first, paying each tranche from the available cash_amount. **Hint**: Ask each tranche how much interest is owed, make the payment, subtract the paid amount from the availableFunds, and move on to the next tranche.
✦ Assuming there is cash left over after all the interest payments, begin cycling through the tranches to make principal payments. There are two separate approaches for this, depending on the value of the *mode* flag on the **StructuredSecurity** object:

- **Sequential**: Cycle through each tranche (in order of subordination), making the maximum principal payment. **Hint**: Ask each tranche for its balance and pay as much as possible based on available cash. If there is cash leftover after paying a tranche (which implies the tranche is fully paid off), use it to start paying off the next tranche.
- **Pro Rata:** Each tranche has a percent of the total notional. Use those percentages to allocate the leftover cash to each one. For example, if leftover cash is $1,000,000, Tranche A is 80%, and Tranche B is 20%, give Tranche A $800,000 and Tranche B $200,000. Of course, be sure to never overpay a tranche's balance (keep the extra cash).

✦ Once all interest and principal payments have been made, it's possible that there is leftover cash. This extra cash goes into a *reserve account*. This is stored in the **StructuredSecurity** object, and should be used to supplement the *cash_amount* at the next, and each following payment. Note that in practice, the reserve account will earn interest, but we will assume 0% interest for simplicity

○ Create a function called **getWaterfall** that returns a list of lists. Each inner list represents a tranche, and contains the following values for a given time period: **Interest Due**, **Interest Paid**, **Interest Shortfall**, **Principal Paid, Balance.**

4. Create a standalone function called **doWaterfall**. This function should take two parameters: A **LoanPool** object and a **StructuredSecurities** object. The function should loop through time periods, starting from 0, and keep going until the **LoanPool** has no more active loans (no more payments coming from the **LoanPool**). At each time period iteration:

- Increase the time period on the **StructuredSecurities** object (which will, in turn, increase for all the tranches).
- Ask the **LoanPool** for its total payment for the current time period.
- Pay the **StructuredSecurities** with the amount provided by the **LoanPool**.
- Call **getWaterfall** on both the **LoanPool** and **StructuredSecurities** objects and save the info into two variables.

After the loop completes (when all loans in the loan pool have zero balance), return all the results saved down from the **getWaterfall** function (from both variables) as well as any amount left in the reserve account.

Test out your code in detail. The best approach is to test out each class and function individually, to ensure they word as expected (with a single or small number of loans). You can do this in **main** or using the Python Console. Once you are confident your classes are correct, try the following, in your **main**:

1. Create a **LoanPool** object that consists of 1,500 loans. Use the provided CSV file of loan data to create these Loan objects.

2. Instantiate your **StructuredSecurities** object, specify the total notional (from the **LoanPool**), add two standard tranches (class A and class B in terms of subordination), and specify sequential or pro-rata mode.

The rates for each tranche can be arbitrary (for now). Note that subordinated tranches should always have a higher rate, as they have increased risk.

3. Call **doWaterfall** and save the results into two CSV files (one for the asset side and one for the liabilities side). All the tranches' data for a given time period should be a single row in the CSV. The reserve account balance should also be in liabilities CSV, for each time period. Each time period gets its own row. Note that you may need to do some clever list comprehensions and string parsing to get this to output correctly.

The goal at this point is to see what happens when running the code for different combinations of tranche input rates: Did each tranche's balance get successfully paid down to 0? If they did, was there any money left in the reserve account at the end?

Congratulations, you've implemented your very first ABS Waterfall! This is the first, and most important, step to valuating ABSs.

Part 2: Waterfall Metrics

Now that we've created the Waterfall, the next step is to be able analyze it. There are a number of metrics that we (or an investor) can use to perform this analysis:

1. 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):

$$0 = -C_0 + \sum_{t=1}^{T} \frac{C_t}{(1+r)^t} \quad \xrightarrow{yields} \quad \boldsymbol{IRR}$$

There is no closed formula to calculate this, since it involves solving the above polynomial equation. There are numerical techniques to calculating IRR, but we will avoid that here by using the **numpy.IRR** function.

2. Average Life (AL): When playing around with your Waterfall code, you may have noticed that some tranches get paid down (0 balance) quicker than others, while some never get fully paid down at all. The AL of the security is the average time that each dollar of its unpaid principal remains unpaid. This is a vital metric for investors, to get a sense of how long it will take them to recoup their principal (will they get most of it back early on in the waterfall, or will it take closer to the end)? In the case when a tranche is never fully paid down, the AL is infinite (no investor would want this security!).

3. 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.

For this part, you should implement three new methods in the **Tranche** base class: **IRR**, **AL**, and **DIRR**:

4. IRR: This should simply delegate to the **numpy.IRR** function. You will need to pass in a list with the initial investment's notional as a negative number, and the total payments for each time period (in order). For example, for a four time period structure with notional 100: **[**-100, 25, 15, 50, 10**]**. Multiply the result by 12 to annualize.
5. DIRR: Simply subtract the IRR from tranche's rate.
6. AL: This is the inner product of the time period numbers (1, 2, 3, etc.) and the outstanding balances, divided by the initial principal. For example, if you have the outstanding balance list as follows: [10000, 90000, 35000, 0], the AL would be (0*-100000+1*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**.
Hint: Use the built-in Python **reduce** function.

Now that you've implemented these metrics, modify the **doWaterfall** function to return the metrics after the Waterfall completes. Use the table from **Appendix A** to translate a DIRR value to a letter rating. Your main function should now output all these metrics and letter rating to the screen (in addition to creating the CSV files from earlier).

Part 3: Valuing the Structure

We've implemented the Waterfall and metrics which will help the investors understand the deal. However, those metrics were based on a fatal assumption: The default/prepayment risk being non-existent! To properly value a structured security, one must properly model expected loss due to defaults and prepayments. For this part, for simplicity, we will include defaults but not prepayments (though prepayments are a fact of life).

Note that most defaults happen somewhere between the early and late time periods (it's rare to default right away or at the very end).

Until now, we've been arbitrarily choosing the tranche interest rates. However, the entire goal of structuring these securities is to come up with *fair* rates. Fair rates are rates that adequately compensate the investors for the risk level of the tranche (which is determined from the DIRR->Rating which is, in turn, determined by the default curve), but are not higher than they should be either. Implementing code to find these rates (and their associated DIRR->Rating) is the objective of this Part. The goal is to find the set of input rates to the tranches that result in a yield of the same rate, on average. If you arbitrarily choose rates, it is almost certain that the Waterfall will result in yields that are either better or worse than the chosen rates (due to the default curve) – which implies that the deal was incorrectly valuated.

If we would know the exact default curve with 100% certainty, then it would be really simple to value the structure: Just incorporate the exact number of loan defaults into the Waterfall and find the tranche rate that results in an actual yield of the same amount. However, we *do not* ever know with *any* certainty what the default curve would look like in real life. Therefore, we need to run a Monte Carlo simulation with thousands of scenarios to simulate different default curves.

In actuality, we need to run two Monte Carlo simulations: The 'inner' simulation which tries thousands of default scenarios and gives the average DIRR and WAL across all the scenarios. Then, based on this result, the 'outer' simulation calculates the yield, from a specified yield curve (using the DIRR and WAL from the 'inner' simulation), tweaks the tranche rates to reflect the new yield, and reruns the inner simulation. The outer simulation keeps doing this until two consecutive runs result in very similar yields (i.e., within a 50 basis point (.005) tolerance).

This all may sound really confusing, but the below steps should help clarify exactly what needs to be done in code:

First, we need to implement loan defaults into the Waterfall. The topic of constructing and using loan default/loss curves is for an entire Masters level course. Therefore, for the purpose of this project, we will be taking an extremely simplistic view of the default curve (which is certainly incorrect in the real world).

1. Use the following table to determine the probability of default of a loan for a given time period:

Time Period Range	Default Probability
1-10	0.0005
11-59	0.001
60-120	0.002
120-180	0.004
180-210	0.002
210-360	0.001

2. Create a method in the **Loan** (called **checkDefault**) class that determines whether or not the loan defaults. The parameter to this method should be a number (see step 3 and 4).

3. Create a method in **LoanPool** (called **checkDefaults**) that generates a list of uniform random integers, one for each **Loan.** The integer range should be so that the odds of a given number occurring is the same odds as the default probability for the time period (always start the range from zero). This method should then call **checkDefault** on each Loan, passing in the random number.

4. checkDefault on each **Loan** flags the loan as defaulted if the passed-in number is 0 (and should return the recovery value of the **Asset**). If the loan is defaulted, a flag should be set on the object and the balance becomes 0.

5. checkDefaults on the **LoanPool** should be called at each **doWaterfall** iteration (time period). It should aggregate the recovery values returned by the individual loans, and return that to **doWaterfall** (to be used as part of the time period's cashflow).

Run your Waterfall (still choosing arbitrary tranche rates) and notice that each run now results in a different DIRR and WAL for a given set of rates; this is because you've incorporated a random component to the loan default model. This random component will enable you to build the Monte Carlo engine to value the structure.

To create the Monte Carlo, you should follow the following rough steps:

• Create a global function called **simulateWaterfall**. Its input parameters should be a **LoanPool**, a **StructuredSecurities** object, and *NSIM*. NSIM specifies the number of simulations to run, and will be used to tell your loop how many times to iterate.

- All this function needs to do is call **doWaterfall** NSIM number of times, and collect the resulting DIRR and AL from each iteration. Once NSIM iterations are complete, take the average of all the DIRR and AL values. Note that if AL is ever infinite (due to the tranche not being paid down), you will need to exclude those AL values from the average.
- The function should return the average DIRR and WAL values for each tranche.

Test this function, first with a low number of simulations to ensure it works, then with 2,000 simulations.

• Create a global function call **runMonte**. Its input parameters should be a **LoanPool**, a **StructuredSecurities** object, *tolerance*, and *NSIM*. The tranches in the **StructuredSecurities** object should be initialized with rates of 5% and 8% for class A and class B respectively. The **runMonte** function should do the following:

- Kick off an infinite loop. The below steps belong inside this infinite loop.

- Invoke the **simulateWaterfall** with the **LoanPool** and **StructuredSecurities** objects, as well as the NSIM. Save down the resulting average DIRR/WAL for each tranche.
- Now that you have the average DIRR and WAL for each tranche, the next step is to calculate the yield. Yield is determined from a *yield curve*. Constructing a proper yield curve is itself a major topic, which is beyond the scope of a Python course. Therefore, all you need to do is implement the below equation (which is one possible yield curve) into a **calculateYield** function. Call this function with the average DIRR/WAL for both tranches and save down the yields:

$$yield = \frac{\left(\frac{7}{1 + .08 * e^{-.19\left(\frac{a}{12}\right)}} + .019\sqrt{\left(\frac{a}{12}\right)(d * 100)}\right)}{100}$$

$$\textbf{where } a \text{ is the } WAL \text{ and } d \text{ is the } DIRR.$$

- Calculate the new tranche rates using the previous tranche rates and the above yield for each tranche. We will use something call *relaxation*, which is a technique to speed up convergence. To achieve this, use the following formula for each tranche:

$$newTrancheRate = oldTrancheRate + coeff * (yield - oldTrancheRate)$$

$$\textbf{where } coeff \text{ is } 1.2 \text{ for Tranche } A \text{ and } 0.8 \text{ for Tranche } B$$

- Check if the new tranche rate differs from the previous tranche rate, for each tranche, by more than *tolerance*. Use the following formula to calculate this:

Don't worry if you don't understand why we are doing this – it's the topic of a numerical methods course. The focus here is solely on implementing this sort of equation into Python.

$$diff = \frac{\left(nA * \left|\frac{lastARate - newARate}{lastARate}\right| + nB * \left|\frac{lastBRate - newBRate}{lastBRate}\right|\right)}{N}$$

$$\textbf{where}:$$
$$nA \text{ is Tranche } A \text{ notional},$$
$$nB \text{ is Tranche } B \text{ notional},$$
$$N \text{ is the total notional.}$$

If *diff* is lower than *tolerance*, then break from the infinite loop – your Monte Carlo is finished! In this case, return the final average DIRR/WAL and yields for each tranche. If *diff* is greater than *tolerance*, then modify the tranche rates to reflect the yields and loop again. Your **main** code should create the **LoanPool** and **StructuredSecurities** object and invoke **runMonte** with those as arguments, in addition to a *tolerance* of 0.005 and *NSIM* of 2000. Once you receive the results, translate the DIRR to a letter rating and output the DIRR, Rating, WAL, and rate of each tranche.

Note that this simulation can take a *really* long time to complete. Multiprocessing can be your savior here. It will be in your interest to modify the above steps to work with multiprocessing, as the simulation will take much, much quicker. The rough steps to do so are below:

1. We cannot parallelize the outer loop (the **runMonte** function). This is because each loop is dependent on the previous. However, we *can* parallelize the **runSimulation** function, since each loop is independent from all the others. To do this, create a function called **runSimulationParallel**. Its input parameters should be a **LoanPool**, a **StructuredSecurities** object, *NSIM* and *numProcesses*. Modify the **runMonte** function to call **runSimulationParallel** instead of **runSimulation**. You should specify 20 for *numProcesses*, but you can experiment with more or less later.

2. runSimulationParallel should start the processes. It should use two Queues, as demonstrated in the Concurrency section of this course. The process' targets should be **doWork** (which monitors the input queue).

- Add *numProcesses* number of tuples to the input Queue. The NSIM argument for **runSimulation** should be *NSIM* divided by *numProcesses*. This makes sure that we evenly split the total simulations between processes.
- The sub-processes monitor the input Queue.
- The main process monitors the output Queue. There is no need to write 'Done' or check for 'Done' in this case; the output Queue monitoring loop can simply exit as soon as the length of the results list is the same as *NSIM*.
- Once the output Queue monitoring loop completes, aggregate and average the resulting tranche metrics from each process and return these to **runMonte.**
- Before returning to **runMonte**, be sure to terminate the sub-processes by calling **stop**() on each sub-process handle.

Your entire simulation should now run significantly quicker than before (maybe even 20x quicker!). Feel free to play around with different numbers of processes to get a sense of the optimal number.

Appendix A

ABS Rating Table

Letter Rating	DIRR (BPS)
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	2,500
Ca	10,000

Source: Rutledge, Raynes (2010)

The DIRR is quoted in *basis points* (BPS). One basis point is 1/100 of a percent. 10,000 basis points is 100% (which is rated **Ca** in the below table). You may need to convert your DIRR to basis points to match.

Note that the above table is one possible example of a ratings table; in practice, these numbers may differ.

Bibliography

Allman, Keith A. *Modeling Structured Finance Cash Flows with Microsoft Excel: a Step-by-Step*. John Wiley & Sons, 2007.

Rutledge, Ann, and Sylvain Raynes. *Elements of Structured Finance*. Oxford Univ. Press, 2010.