# SIMPLEX

**SYNTAX FOR INTERNATIONAL MONETARY, PROPERTY, AND LIQUIDITY EXCHANGE**

Steven Chen Gilbert Hom Kelvin Jiang Eric Zhang
{stc2104, gch2102, kxj1, ehz2101}@columbia.edu

## INTRODUCTION

The financial services industry is dynamic and extremely competitive. From making investment choices to providing advice to Fortune 500 corporations, one's reputation depends on the ability to make wise, precise and timely decisions. SIMPLEX is designed to assist financial professionals quickly produce accurate and precise financial analysis under intense deadline pressure.

SIMPLEX allows financial professionals to focus on analysis rather than on intricacies in programming languages such as Java and C++. The language is syntactically intuitive and uses terminology and conventions familiar to professionals in the industry. Furthermore, users of SIMPLEX will appreciate the built-in features which again, allow the user to focus on the analysis and not on formatting output, memorizing formulas, determining data types or learning a significant amount of syntax. Overall, SIMPLEX is a flexible and intuitive programming language which can be used across many platforms and serves as a powerful tool for financial professionals.

## BACKGROUND

Financial models serve as the foundation upon which financial analysis occurs. Financial models are used to calculate and/or estimate financial figures. They can be used to predict future earnings of a company, the result of a potential merger/acquisition, and the amount of debt a company can take on, among other things. Good models are designed in such a fashion that, should a user change an initial assumption, the change in the assumption should flow through the model without having to update anything else. Furthermore, good models can be easily modified and expanded to provide greater functionality and flexibility.

## MOTIVATING SCENARIOS

An investment banking analyst has been asked to determine the pro forma effect of the merger between two large media companies. A homeowner wants to determine the best mortgage that fits his/her financial situation. A student wants to keep track of her student loans and the payments needed to repay those loans. An entrepreneur needs to estimate the growth of the sales of his company for a presentation to potential investors. An economist wants to determine the effect of a rise in interest rates by the Fed. A foreign exchange trader wants exploit currency arbitrage situations. These are all scenarios in which SIMPLEX will be able to provide a path to an answer or a solution.

While Microsoft Excel is powerful and considered the industry standard when it comes to financial analysis, it is fairly expensive. For students and young entrepreneurs who want to perform their own financial analysis, Microsoft Excel is most likely unaffordable. SIMPLEX will give its users the power to create everything from expansive financial models to quick financial calculations at virtually no cost. Overall, SIMPLEX serves as an economical alternative to Microsoft Excel without compromising any features required to generate meaningful financial analysis.

## FUNDAMENTAL FEATURES

### *Data Types*

A significant feature of SIMPLEX is the use of different currencies as data types. For example, US Dollars, Euros, Japanese Yen and other currencies will be set as data types. The reason motivating this feature is to ensure that different currencies cannot be mathematically manipulated without first applying the appropriate currency exchange rate. This ensures apples-to-apples comparisons. In addition SIMPLEX will include a data type called 'rate.' Users will program rates—such as interest rates—as 7.50 rather than 0.075. It is more intuitive to think of rates in terms of their actual number (7.50%) rather than their decimal equivalent (0.075).

In order to keep track of time, SIMPLEX will support 'years,' 'months' and 'days' data types. Lastly, for all other purposes, the programming language will have a generic string and number data type.

### *Currency Manipulation*

SIMPLEX will automatically import currency exchange rate data in order to perform manipulations between currencies. This is a valuable feature because it prevents the user from having to enter all applicable exchange rate ratios into his/her program. A unique feature of SIMPLEX will include support to quickly typecast currencies. For example, if it was necessary to convert a value from US Dollars to Euros, the user could simply type `(Euros)` in front of a US Dollar variable name. This automatically applies the appropriate exchange rate to the value of the variable being converted.

### *Appropriately Formatted Output*

In order to differentiate between currencies when values are displayed, the accompanying symbol (e.g. $, ¥, €, £, etc.) is also printed. Rather than having a minus sign in front of negative values, which are often small and hard to see, negative values will be shown with parenthesis around them. Also, values being printed will always have the appropriate amount of significant digits shown. For example, US Dollars are represented as having 2 decimal places and never more. This feature ensures that no superfluous data is shown.

### *Built-In Functions*

Due to the mathematical complexity of some formulas, SIMPLEX will contain several frequently used functions including payment, present value, future value and internal rate of return functions.

### *Mathematical Expressions*

SIMPLEX offers the convenience of being able to perform and evaluate mathematical expressions through the use of operators, not procedures and functions. This mimics the format of evaluating mathematical expressions in graphical calculators and Microsoft Excel. Having this feature focuses the user's attention to the program and not syntax. For example `Math.pow(3,2)` is the Java operation to raise 3 to the second power. The syntax to perform the same operation in SIMPLEX is `2^3`.

*User-Defined Functions*

A rudimentary equations solver will be incorporated into the compiler. Users can use this solver to define functions which take parameters of linear equations and solve for the missing variable. This way, long and common strings of arithmetic can be condensed into a user defined function. The user can then call the function in the program, leaving one of the arguments undefined, and the function will return the value of the missing variable.

*Portability*

An essential component of SIMPLEX is its ability to be run on several computing platforms. To achieve this feature, SIMPLEX code is translated into Java, which is then able to run on computers with the Java Virtual Machine installed.

## SYNTAX SAMPLE

```
// Mortgage Calculator written in SIMPLEX

// Declaration of Variables
USD totalMortgage;
years numYears;
number paymentsPerYear;
rate interestRate;
USD payment;

// Get Inputs from User
Input(“Enter Total Mortgage amount: “, totalMortgage);
Input(“Enter interest rate: “, interestRate);
Input(“Enter Payments Per Year: “, paymentsPerYear);
Input(“Enter Number of years: “, numYears);

// Calculate Payment manually
payment = totalMortgage * (interestRate / paymentsPerYear);
payment = payment * ((1+interestRate)^(numYears*paymentsPerYear));
payment = payment / ((1+interestRate)^(numYears*paymentsPerYear)-1);

// calculate and output amortization schedule
number currentPeriod = 0;
USD beginningBalance = totalMortgage;
USD endingBalance;
USD interestPayment;
USD amort;

print ("PERIOD  BEGINNING BALANCE  INTEREST PAYMENT  AMORTIZATION  ENDING BALANCE");
print ("------  -----------------  ----------------  ------------  --------------");

while (currentPeriod >= (numYears*paymentsPerYear))
{
       interestPayment = (beginningBalance * (interestRate / paymentsPerPeriod));
       amort = (payment - interestPayment);
       endingBalance = (beginningBalance - amort);

       print (period, "  ", beginningBalance, "  ", interestPayment, "  ", amort, "  ",
endingBalance);

       beginningBalance = endingBalance;
       currentPeriod++;

}

// end of program
```

## JAVA IMPLEMENTATION

```
import java.lang.Math;

// Mortgage Calculator written in JAVA

public class MortgageCalculator
{
    // instance variables
    double amount;
    int years;
    int paymentsPerYear;
    double interestRate;
    int numberOfPeriods;
    double payment;
    double interestRatePerPeriod;

    /**
     * Constructor for objects of class MortgageCalculator
     */
    public MortgageCalculator(double amount, int years, int paymentsPerYear, double interestRate)
    {
        this.amount = amount;
        this.years  = years;
        this.paymentsPerYear = paymentsPerYear;
        this.interestRate = interestRate;

        numberOfPeriods = years * paymentsPerYear;
        interestRatePerPeriod = (interestRate / (double)paymentsPerYear);
        interestRatePerPeriod = interestRatePerPeriod / 100;
    }

    public double calcPayment()
    {
        // calc payment based on number of payments per year (formula)

        double paymentNumerator, paymentDenominator;

        paymentNumerator = amount * interestRatePerPeriod;
        paymentNumerator = paymentNumerator * (Math.pow((1 + interestRatePerPeriod),
numberOfPeriods));

        paymentDenominator = (Math.pow((1 + interestRatePerPeriod), numberOfPeriods)) -1;

        payment = (paymentNumerator / paymentDenominator);
        payment = roundCurrency(payment);

        return payment;

    }

    public void printPerPeriod()
    {
        int counter;
        double balance;
        double interestPayment;
        double amort;

        balance = amount;
        roundCurrency(balance);


        for (counter = 1; counter <= numberOfPeriods; counter++)
        {
            System.out.print("Period " + counter + "\t");

            System.out.print("$" + balance + "\t");
            System.out.print("$" + payment + "\t");

            interestPayment = roundCurrency(interestRatePerPeriod * balance);
            System.out.print("$" + interestPayment + "\t");

            amort = roundCurrency(payment - interestPayment);
            System.out.print("$" + amort + "\t");

            balance = roundCurrency(balance - amort);
            System.out.print("$" + balance + "\n");

        }
    }

    public double roundCurrency(double in)
    {
        double out;
        long temp;

        // multiply by 100, round, then divide by 100 again
        in = in * 100;
        temp = Math.round(in);
        out = ((double)temp / 100);

        return out;

    }

}
```