# MQF 633 C++ for Financial Engineering

# Lecture 10: Final Project - Derivative Portfolio P&L and Risk

### Common Mistakes

#### IDE & Env related

* Having “space” in your path holding the IDE: for windows it might be ok, but for Linux related OS, this is definitely not OK. There will be error of cannot find file of path etc.
* Having 2 versions of same compiler installed
* Using wrong settings, ${file} vs ${workpathFolder}\\*.cpp, only one .exe for windows and one executable files for mac (main.cpp).

#### Git related

* Check in everything: OS, IDE related setting there is “NO NEED” to check-in. Just check in source file (.h, .cpp) and your result file or input file (txt. csv)

#### C++ syntax related

* Trying to include all .h files in main.cpp - Principal is to keep include minimum, only
  + #include into immediate .cpp files
  + if you can include into .cpp file to make it work, then do not include into .h files
* Put function implementation in .h file but did not using inline. Having error of duplicated symbols during compiling time. Remember, function body (implementation) should not be copied twice into one compiling unit scope.

#### Math related

* Long bond has -Ve NPV? PV - Present value is market value of an asset.
* Binomial tree time step is not enough. To have better converge to Black Scholes, consider using daily or at least weekly for a one-year option.

### Final Project Requirement

Create a console application that value a set of derivatives trades, compute the PV and Greeks.

#### Some key notes:

Market data:

1. Load curve data and create IR curve of USD-SORF and SGD-SORA:
   1. IR curve should have bucket tenor as Date class
   2. Rate we use data provided directly as zero coupon rate
   3. Interpolation function to get zero rate for any given date from curve
   4. Choose DF = - exp (Zr x T), Zr is zero rate from curve, and T is the date required to price derivative
2. Load vol data, create vol curve of tenor and vol value
   1. Vol data as lognormal vol
   2. Interpolation function which calculates vol using expiry date from vol curve

Trade:

1. Load trade from trade data file provide, create a portfolio of trades
2. Implement necessary PV function or Price function for each trade class

Model:

Swap and bond:

1. PV can be computed as discounting of future cash flows, using discount factor interpolated from IR curve
2. For swap, it can be also computed as (fixed leg rate – par rate) \* annuity. Par rate can be approximately interpolated from IR curve using swap end date, and annuity = fix leg notional \* sum of (coupon period \* discount factor at each coupon payment date of fixed leg).

European:

1. Price with black model
2. Price with any Binomial tree model, using time step = 50
3. Spot price should be taken from stock price file and IR rate and vol should be interpolated from given IR curve or Vol curve using expiry date

American:

1. Price with any Binomial tree model, using time step = 50
2. Spot price should be taken from stock price file and IR rate and vol should be interpolated from given IR curve or Vol curve using expiry date

Risk Engine:

1. Implement the risk engine to compute Delta of IR and Vega of vol
2. Central difference method to bump and IR curve (1bp) and vol Curve (1%) in parallel and compute the sensitivity as change of PV.

## Implementation requirements

1. No raw pointer should be used
2. Use at least one of design patterns

## Result requirement

1. Output PV, and Greek (total delta and total vega) by trade into a txt file.
2. A short write-up to summarize observation and add your explanation
   1. Comparing tree model price of European vs Black, and explain the difference
   2. Comparing the American and European trade pv and explain the difference

### Some Helps and Sample Code

1. Curve interpolator

We can write a simple linear interpolation function to get zero rate from curve.

* Assuming our curve are interpolated zero coupon rate curve;
* Assuming the tenor dates are sorted ascending order;

namespace imp {

double linearInterpolate(double x0, double y0, double x1, double y1, double x)

{

if (x < x0)

return y0;

else if (x > x1)

return y1;

else

return y0 + (x - x0) \* (y1 - y0) / (x1 - x0);

}

}

double RateCurve::getRate(Date date) const

{

//use linear interpolation to get rate

auto it = std::lower\_bound(tenors.begin(), tenors.end(), date);

if (it == tenors.end()) // cannot find any item which is >= value

return \*rates.end();

if (it == tenors.begin())

return \*rates.begin();

Date dt = \*it;

if (dt == date)

return rates[it - tenors.begin()];

else {

auto it0 = it;

it0--;

double x0 = it0->getSerialDate();

double y0 = rates[it0 - tenors.begin()];

double x1 = it->getSerialDate();

double y1 = rates[it - tenors.begin()];

double x = date.getSerialDate();

double newRate = imp::linearInterpolate(x0, y0, x1, y1, x);

return newRate;

}

}

Similar function can be used to interpolate vol from expiry of trades as well.

1. Schedule generation for Bond and Swap

Use the function below or similar to generate schedule of dates for Bond and Swap from 1st accrual start date to the last accrual end date. Below sample code is for swap.

long Date::getSerialDate() const

{ //1900-1-1 ->1

// Adjust for Excel's incorrect leap year handling (Excel treats 1900 as a leap year)

int daysSinceEpoch = 0;

for (int y = 1900; y < year; ++y) {

daysSinceEpoch += (y % 4 == 0 && (y % 100 != 0 || y % 400 == 0)) ? 366 : 365;

}

int daysInMonth[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

if (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)) {

daysInMonth[1] = 29; // Leap year adjustment

}

for (int m = 0; m < month - 1; ++m) {

daysSinceEpoch += daysInMonth[m];

}

daysSinceEpoch += day;

// Excel incorrectly considers 1900 as a leap year, so we add 1 for compatibility

if (year > 1900) {

daysSinceEpoch += 1;

}

return daysSinceEpoch;

}

void Date::serialToDate(int serial)

{

int daysSinceEpoch = serial-2; // Adjust for Excel's incorrect leap year handling

int y = 1900;

int m = 1;

int d = 1;

while (daysSinceEpoch > 0) {

int daysInMonth;

switch (m) {

case 4: case 6: case 9: case 11: daysInMonth = 30; break;

case 2: daysInMonth = (y % 4 == 0 && (y % 100 != 0 || y % 400 == 0)) ? 29 : 28; break;

default: daysInMonth = 31;

}

if (daysSinceEpoch >= daysInMonth) {

daysSinceEpoch -= daysInMonth;

m++;

if (m > 12) {

m = 1;

y++;

}

}

else {

d += daysSinceEpoch;

break;

}

}

year = y;

month = m;

day = d;

serialNumber = serial;

}

Date dateAddTenor(const Date& start, const string& tenorStr)

{

Date newdate;

if (to\_lower(tenorStr) == "on" || to\_lower(tenorStr) == "o/n") {

long newSerial = start.getSerialDate() + 1;

newdate.serialToDate(newSerial);

}

else {

int numUnit = stoi(tenorStr.substr(0, tenorStr.size() - 1));

auto tenorUnit = tenorStr.back();

if (tenorUnit == 'W') {

long newSerial = start.getSerialDate() + numUnit \* 7;

newdate.serialToDate(newSerial);

}

else if (tenorUnit == 'M') {

long newSerial = start.getSerialDate() + numUnit \* 30;

newdate.serialToDate(newSerial);

}

else if (tenorUnit == 'Y') {

long newSerial = start.getSerialDate() + numUnit \* 360;

newdate.serialToDate(newSerial);

}

else

throw std::runtime\_error("Error: found unsupported tenor: " + tenorStr);

}

return newdate;

}

void Swap::generateSwapSchedule()

{

if (startDate == maturityDate || frequency <= 0 || frequency > 1)

throw std::runtime\_error("Error: start date is later than end date, or invalid frequency!");

string tenorStr;

if (frequency == 0.25)

tenorStr = "3M";

else if (frequency == 0.5)

tenorStr = "6M";

else

tenorStr = "1Y";

Date seed = startDate;

while (seed < maturityDate) {

swapSchedule.push\_back(seed);

seed = dateAddTenor(seed, tenorStr);

}

swapSchedule.push\_back(maturityDate);

if (swapSchedule.size() < 2)

throw std::runtime\_error("Error: invalid schedule, check input!");

}

1. Bond PV to be priced by using cash flow discounting from IR curve

Imagin the case of holding long bond position with notional of N, coupon rate of C, maturity in m year and with frequency of f per annum, then we have PV of bond from cash flow discounting:

* We will have K = m x f number of coupon periods.
* Sign = 1 since the position is long, if short position then sign = -1;

PV = sign x Sum of (C x DF(i) x YF(i) + N x DF(k), where i = 0 to K, YF(i) is coupon period in year fraction for each period.

Then we need to compute all the DF(i) from curve by interpolating Zr(i) from each coupon payment date. Then DF = exp(-Zr x T) will give you the DF for each date.

1. Swap PV from cash flow discounting

In general, the formula below holds:

NPV = Floating Leg PV + Fixed Leg PV, and

Floating Leg PV = Notional - Notional x Discount Factor (at end date of floating leg)

Fixed leg PV = Sum of (coupon(i) x DF(i) at each payment date of fixed leg)

Coupon of each period = fixed leg Notional x coupon rate of fixed leg x year fraction of each coupon period

\* Notional is leg notional need to reflect pay and receive using - / +

Method 1: to compute DF(i) at each payment date and then compute the PV

Method 2: using NPV = fixed leg annuity x (trade rate – par rate) where

* Trade rate is swap rate on fixed leg
* Par rate is computed as below

Par rate = floating leg PV / annuity of fix leg and

Annuity of fix leg = sum of (N \* coupon period(i) \* DF (i))

Below sample code for annuity of swap.

class Swap : public Trade {

public:

//make necessary change

Swap(string name, Date start, Date end, double \_notional, double \_rate, double \_freq)

{

tradeType = "Swap";

underlying = name;

startDate = start;

maturityDate = end;

tradeDate = start;

notional = \_notional;

tradeRate = \_rate;

frequency = \_freq;

generateSwapSchedule();

}

/\*

implement this, using npv = discounted cash flow from both leg;

\*/

inline string getType() const { return tradeType; };

inline string getUnderlying() const { return underlying; };

double Payoff(double r) const;

double Pv(const Market& mkt) const;

double getAnnuity() const; //implement this in a cpp file

void generateSwapSchedule();

private:

Date startDate;

Date maturityDate;

double tradeRate; // fixed leg rate

double frequency; // use 1 for annual, 2 for semi-annual etc

vector<Date> swapSchedule;

};

double Swap::getAnnuity(const Market& mkt) const

{

double annuity = 0;

Date valueDate = mkt.asOf;

auto rc = mkt.getCurve(rateCurve);

for (size\_t i = 1; i < swapSchedule.size(); i++) {

auto dt = swapSchedule[i];

if (dt < valueDate)

continue;

double tau = (swapSchedule[i] - swapSchedule[i - 1]) / 360;

double df = rc->getDf(dt);

annuity += notional \* tau \* df;

}

return annuity;

}

double Swap::Pv(const Market& mkt) const

{

//using cash flow discunting

Date valueDate = mkt.asOf;

auto rc = mkt.getCurve(rateCurve);

double df = rc->getDf(maturityDate);

double fltPv = (-notional + notional \* df);

double fixPv = 0;

for (size\_t i = 1; i < swapSchedule.size(); i++) {

auto dt = swapSchedule[i];

if (dt < valueDate)

continue;

double tau = (swapSchedule[i] - swapSchedule[i - 1]) / 360;

df = rc->getDf(dt);

fixPv += notional \* tau \* tradeRate \* df;

}

return fixPv + fltPv;

}

Option PV

Use PV coming from one of Binomial tree model

### Risk Computation

1. Swap & Bond DV01 – NPV sensitivity of IR curve bumps

DV01 represents the interest rate risk for Swap and Bond on IR curve. Since now we are computing Swap and Bond NPV using cash flow discounting method, it is natural we can compute the **DV01 = dNPV/dZr**. Take note this is a vector of sensitivity for each curve, with all the buckets which leads to none zero sensitivities.

Here we compute DV01 using below setup:

* Set curve used for each deal of swap or bond
* Parallel shock all tenors rate by 1bp
* Calculation total DV01 per deal by central difference: DV01 = dNPV/dZr = (NPV(bump up) - NPV(bump down)) / 2

So the key idea here is to bump the market data and re-price the trade. It is quite important that since Market is the shared object, trying to ensure original market data is not being changed will generally give the ideal behavior of thread-safe in MT environment.

1. Option deals DV01

It can be computed similarly since they are also using IR curve for discounting

1. Vega risk of European Option and American Option

Vega = dNPV/dVol

Shock size: 1%, to be applied to Vol curve

Calcualtion: central difference, dNPV/dVol = NPV+(bump up) - NPV- (bump down) / 2

Code Implementation

Define struct of Market data shock as below, to present the shocked market data point.

struct MarketShock {

string market\_id;

pair<Date, double> shock; //tenor and value

};

Define market data mutator using design pattern of decorator. Market data mutator can be viewed as the “shocked” version of original market data. It contains bumped up version and bumped down version as private member. Below example shows curve mutator where ir curve got bumped. At curve class, we need to implement shock() function.

class CurveDecorator : public Market {

public:

CurveDecorator(const Market& mkt, const MarketShock& curveShock) : thisMarketUp(mkt), thisMarketDown(mkt)

{

cout << "curve decorator is created" << endl;

auto curve\_up = thisMarketUp.getCurve(curveShock.market\_id);

curve\_up->shock(curveShock.shock.first, curveShock.shock.second);

cout << "curve name " << curveShock.shock.first << "is shocked" << curveShock.shock.second << endl;

auto curve\_down = thisMarketDown.getCurve(curveShock.market\_id);

curve\_down->shock(curveShock.shock.first, -1 \* curveShock.shock.second);

cout << "curve name " << curveShock.shock.first << "is shocked" << curveShock.shock.second << endl;

}

inline const Market getMarketUp() const { return thisMarketUp; }

inline const Market getMarketDown() const { return thisMarketDown; }

private:

Market thisMarketUp;

Market thisMarketDown;

};

void RateCurve::shock(Date tenor, double value)

{

// parallel shock all tenors rate

for (auto& rt : rates) {

rt += value;

}

}

**Pay attention** to the Market data copy construction, since now market data is having pointer of curve and vol as its member, we need to have deep copy behaviour so that the bumped IR curve and vol curve is not the original IR curve and Vol curve, but a new copy from it.

Market(const Market& other) {

this->asOf = other.asOf;

// Deep copy each Curve

for (const auto& curve : other.curves) {

curves.emplace(curve.first,std::make\_shared<RateCurve>(\*curve.second)); // Deep copy each Curve

}

// Deep copy each Curve

for (const auto& vol : other.vols) {

curves.emplace(vol.first, std::make\_shared<VolCurve>(\*vol.second)); // Deep copy each Curve

}

bondPrices = other.bondPrices;

stockPrices = other.stockPrices;

}

In the same way we can implement vol decorator for vol bump. And for simplicity, we also choose to bump vol curve in parallel.

void VolCurve::shock(Date tenor, double value)

{

// parallel shock all tenors rate

for (auto& v : vols) {

v += value;

}

}

class VolDecorator : public Market {

public:

VolDecorator(const Market& mkt, const MarketShock& volShock) : thisMarket(mkt)

{

cout << "vol decorator is created" << endl;

auto curve = thisMarket.getVolCurve(volShock.market\_id);

curve->shock(volShock.shock.first, volShock.shock.second);

cout << "vol curve " << volShock.shock.first << "is shocked" << volShock.shock.second << endl;

}

inline const Market& getMarket() const { return thisMarket; }

private:

Market thisMarket;

};

#### Risk Engine

Now we put everything together in to Risk Engine. Risk Engine is a class where creates the mutator, ie, bumped market data, and uses them to price the deal to get the PV difference between original market data and bumped market data.

class RiskEngine

{

public:

RiskEngine(const Market& market, double curve\_shock, double vol\_shock, double price\_shock) {

//add implementation, create curve shocks, vol shocks w.r.t to curve structure etc

cout << " risk engine is created .. " << endl;

};

void computeRisk(string riskType, std::shared\_ptr<Trade> trade, bool singleThread);

inline map<string, double> getResult() const {

cout << " risk result: " << endl;

return result;

};

private:

unordered\_map<string, CurveDecorator> curveShocks; //tenor, shock

unordered\_map<string, VolDecorator> volShocks;

unordered\_map<string, PriceDecorator> priceShocks;

map<string, double> result;

We need to implement the compute risk function, and below is example code. It support single thread mode and using std::asycn to have some sort of multi-threading mode. Since all our pv function has const in its modifier, this means that the shared data of market or trade will not be modified by any thread, hence there is no racing condition.

void RiskEngine::computeRisk(string riskType, std::shared\_ptr<Trade> trade, bool singleThread)

{

result.clear();

if (singleThread) {

if (riskType == "dv01") {

for (auto& kv : curveShocks) {

string market\_id = kv.first;

auto mkt\_u = kv.second.getMarketUp();

auto mkt\_d = kv.second.getMarketDown();

double pv\_up = trade->Pv(mkt\_u);

double pv\_down = trade->Pv(mkt\_d);

double dv01 = (pv\_up - pv\_down) / 2.0;

result.emplace(market\_id, dv01);

}

}

if (riskType == "vega") {

for (auto& kv : volShocks) {

// to be added

}

}

if (riskType == "price") {

// to be added

}

}

else {

auto pv\_task = [](shared\_ptr<Trade> trade, string id, const Market& mkt\_up, const Market& mkt\_down) {

double pv\_up = trade->Pv(mkt\_up);

double pv\_down = trade->Pv(mkt\_down);

double dv01 = (pv\_up - pv\_down) / 2.0;

return std::make\_pair(id, dv01);

};

vector<std::future<std::pair<string, double>>> \_futures;

// calling the above function asynchronously and storing the result in future object

for (auto& shock : curveShocks) {

string market\_id = shock.first;

auto mkt\_u = shock.second.getMarketUp();

auto mkt\_d = shock.second.getMarketDown();

\_futures.push\_back(std::async(std::launch::async, pv\_task, trade, market\_id, mkt\_u, mkt\_u));

}

for (auto&& fut : \_futures) {

auto rs = fut.get();

result.emplace(rs);

}

}

}

Trade Creation Using Factory Pattern

Remember to make necessary change in the trade class constructor.

// Abstract creator class

class TradeFactory {

public:

virtual std::shared\_ptr<Trade> createTrade(std::string underlying, Date start, Date end, double notional, double strike, double freq, OptionType opt) = 0;

virtual ~TradeFactory() {} // Virtual destructor for polymorphism

};

// Concrete creator class - SwapFactory

class SwapFactory : public TradeFactory {

public:

std::shared\_ptr<Trade> createTrade(std::string underlying, Date start, Date end, double notional, double strike, double freq, OptionType opt) override {

return std::make\_shared<Swap>(underlying, start, end, notional, strike, freq);//implement this

}

};

// Concrete creator class - BondFactory

class BondFactory : public TradeFactory {

public:

std::shared\_ptr<Trade> createTrade(std::string underlying, Date start, Date end, double notional, double strike, double freq, OptionType opt) {

return std::make\_shared<Bond>(underlying, start, end, notional, strike, freq);// implement this

}

};

// Concrete creator class - EuropeanFactory

class EurOptFactory : public TradeFactory {

public:

std::shared\_ptr<Trade> createTrade(std::string underlying, Date start, Date end, double notional, double strike, double freq, OptionType opt) {

auto trade = make\_shared<EuropeanOption>(opt, notional, strike, start, end);

return trade;

}

};

// Concrete creator class - AmericanOptFactory

class AmericanOptFactory : public TradeFactory {

public:

std::shared\_ptr<Trade> createTrade(std::string underlying, Date start, Date end, double notional, double strike, double freq, OptionType opt) {

return std::make\_shared<AmericanOption>(); //implement this

}

};

Putting Back Everything into Main function

int main()

{

// Get the current system time

auto now = std::chrono::system\_clock::now();

std::time\_t t = std::chrono::system\_clock::to\_time\_t(now);

std::tm localTime;

localtime\_s(&localTime, &t);

Date valueDate = Date(localTime.tm\_year + 1900, localTime.tm\_mon + 1, localTime.tm\_mday);

// step1: create market data and load curve, vol and prices into market data

auto mkt = make\_shared<Market>(valueDate);

loadCurve(\*mkt, "usd\_curve.txt", "USD-SOFR");

loadCurve(\*mkt, "sgd\_curve.txt", "SGD-SORA");

mkt->Print();

auto usdCurve = mkt->getCurve("USD-SOFR");

double rate = usdCurve->getRate(Date(2026, 1, 1));

//task 2, create a portfolio of bond, swap, european option, american option

vector<std::shared\_ptr<Trade>> myPortfolio;

string fileName = "trade.txt";

string header;

vector<string> tradeData;

readFromFile(fileName, header, tradeData);

loadTrade(myPortfolio);

auto bFactory = std::make\_unique<BondFactory>();

auto sFactory = std::make\_unique<SwapFactory>();

auto eFactory = std::make\_unique<EurOptFactory>();

auto aFactory = std::make\_unique<AmericanOptFactory>();

auto bond = bFactory->createTrade("usd-gpv", Date(2024, 1, 1), Date(2034, 1, 1), 1000000, 0.035, 0.6, OptionType::None);

auto swap = sFactory->createTrade("usd-sofr", Date(2024, 1, 1), Date(2034, 1, 1), -1000000, 0.03, 1.0, OptionType::None);

auto eCall = eFactory->createTrade("appl", Date(2024, 1, 1), Date(2025, 1, 1), 10000, 530, 0, OptionType::Call);

auto aPut = aFactory->createTrade("appl", Date(2024, 1, 1), Date(2026, 1, 1), 10000, 525, 0, OptionType::Put);

myPortfolio.push\_back(bond);

myPortfolio.push\_back(swap);

myPortfolio.push\_back(eCall);

myPortfolio.push\_back(aPut);

//task 3, creat a pricer and price the portfolio, output the pricing result of each deal

//3.1 compute the NPV of deal as of market date 1

//3.2 compute the NPV of deal as of market date 2, and then compute the daily Pnl for each deal uisng NPV(date2) - NPV (date1), and output the result in file

auto pricer = new CRRBinomialTreePricer(50);

for (size\_t i = 0; i < myPortfolio.size(); i++) {

auto& trade = myPortfolio[i];

double pv = pricer->Price(\*mkt, trade);

//log pv details out in a file

}

//task 4, compute the Greeks of DV01, and Vega risk as of market date 1

// 4.1 compute risk using risk engine

// 4.2 use idea of multi-threading

// analyzing the pv and risk

// sample code for risk computation

double curve\_shock = 0.0001;// 1 bp of zero rate

double vol\_shock = 0.01; //1% of log normal vol

double price\_shock = 1.0; // shock in abs price of stock

// example 1, simple example of computing one point dv01 for one swap

string risk\_id = "USD-SOFR:DV01:DEAL 01";

double shockUp = 0.0001;

double shockDown = -0.0001;

auto testShockUp = MarketShock();

testShockUp.market\_id = "USD-SOFR";

testShockUp.shock = make\_pair(Date(), shockUp);

auto testShockDown = MarketShock();

testShockDown.market\_id = "usd-sofr";

testShockDown.shock = make\_pair(Date(), shockDown);

auto shockedUpCurveUp = CurveDecorator(\*mkt, testShockUp);

auto shockedUpCurveDown = CurveDecorator(\*mkt, testShockDown);

unordered\_map<string, double> thisDealDv01;

double pv\_up, pv\_down;

auto m\_up = shockedUpCurveUp.getMarketUp();

pv\_up = swap->Pv(m\_up);

auto m\_down = shockedUpCurveDown.getMarketDown();

pv\_down = swap->Pv(m\_down);

double dv01 = (pv\_up - pv\_down) / 2.0;

thisDealDv01.emplace(risk\_id, dv01);

//example2, using risk engine to compute full set of dv01 for a swap

RiskEngine re(\*mkt, curve\_shock, vol\_shock, price\_shock);

re.computeRisk("dv01", swap, true);

auto dv01\_of\_swap = re.getResult();

//example 3, demo using thread pool

ThreadPool pool(4);

map<string, double> swapDv01;

auto pv\_job = [&swapDv01, risk\_id, &swap, &m\_up, &m\_down]() {

cout << "Task is running on thread: " << this\_thread::get\_id() << endl;

auto pricer = std::make\_unique<CRRBinomialTreePricer>(100);

double pv\_u = pricer->Price(m\_up, swap);

double pv\_d = pricer->Price(m\_down, swap);

double dv01 = (pv\_u - pv\_d) / 2.;

swapDv01.emplace(std::make\_pair(risk\_id, dv01));

this\_thread::sleep\_for(chrono::milliseconds(100));

};

for (int i = 0; i < 5; ++i) {

pool.enqueue(pv\_job);

}

//final

cout << "Project build successfully!" << endl;

return 0;

}