# MQF 633 C++ for financial engineering

# Lecture 10: Final Project - Derivative Portfolio PnL and Risk

### Part I: Common Mistake in Mid-Term Project

#### IDE & Env related

* Having “space” in your path holding the IDE: for windows it might be ok, but for Linux related OS, this is defnitely 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 everyhting: 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 incldue all .h files in main.cpp - Princiapl 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 inlcude into .h files
* Put function implementation in .h file but did not using inline. Having error of duplicated symbols. Remeber, funciton body (implmenetation) should not be copied twince into one compling unit scope.

#### Math related

* Long bond has -Ve NPV? PV - Present value is market value of an asset. PnL is profit and lost coming from holding an asset.
* Binomial tree time step is not enough. To have better converge to Black scholse, considering to use daily or at least weekly for a one year option.

### Part II Final Project - Portfolio Risk and PnL

1. Create a portfolio of trades using trade info given in attached portfolio.txt
   1. should not use raw pointer anymore
   2. will be a plus if using design pattern to create trade
2. Create 2 days market data using market data attached.
   1. avoid using raw pointer at all
3. Price the portfolio and output the PV and Risk (DV01 and Vega) for 2 dates
   1. Output the pv and risk result for each date into file
   2. Compute the PnL of each trade using PV difference between 2 dates
   3. Using multi-threading to compute the pv and risk will be a plus
4. Analysing the Greeks of your portfolio and design a small strategy either
   1. Square off the risk of portfolio
   2. Maximise portfolio PnL given a fixed risk limit of Dv01 and Vega

Swap and Bond: NPV to be priced by using cash flow discounting, using provided interest curve.

* **Schedule generation**

this is sample code. this function should be implmented in swap and bond class, to create scheduel as vector<Date> rahter than vector<double>

void inline genSchedule(double start, double end, double freq, vector<double>& schedule)

{

if (start >= end || freq <= 0 || freq > 1)

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

double seed = end;

while(seed > start){

schedule.push\_back(seed);

seed-=freq;

}

schedule.push\_back(start);

if (schedule.size()<2)

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

std::reverse(schedule.begin(), schedule.end());

}

* **Swap NPV**

NPV = Floating Leg PV + Fixed Leg PV

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

Fixed leg pv = Sum of (coupon at each period x Discount Factor (at eaach 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 - / +

* **Bond NPV**

Sum of (coupon at each period x Discount Factor (at eaach payment date of fixed leg) + Notional x Discount Factor (at end date of floating leg)

Coupon of each period = Bond Notional x coupon rate x year fraction of each coupon period

class Swap : public Trade {

public:

//make necessary change

Swap(std::string name, Date start, Date end, double \_notional, double \_rate, double \_freq): Trade("SwapTrade", start)

{

underlying = name;

startDate = start;

maturityDate = end;

notional = \_notional;

tradeRate = \_rate;

frequency = \_freq;

generateSwapSchedule();

}

/\*

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

\*/

double Payoff(double r) const;

double Pv(const Market& mkt) const;

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

void generateSwapSchedule();

private:

string underlying;

Date startDate;

Date maturityDate;

double notional;

double tradeRate;

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

vector<Date> swapSchedule;

};

Option trade NPV

No change, use pv coming from Binomial tree model

Trade PnL Computation

We compute Trade daily PnL = NPV (Date T + 1) - NPV (Date T) for all type of deals

Risk Computation

Swap and Bond DV01

DV01 represent the intererst rate risk for Swap and Bond. Since now we are computing Swap and Bond NPV using cash flow disounting moethod, it is natrual we can compute the DV01 = dNPV/dZr. Take note this is a vector of sensitivty for each curve, with all the buckets which leads to none zero sensitivities.

Shock size: 1bp, to be applied to Zero Coupon rate of curve

Calcualtion: central difference, 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 generatly give the ideal behaivour of thread-safe in MT environment.

struct MarketShock {

string market\_id;

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

};

class CurveDecorator : public Market {

public:

CurveDecorator(const Market& mkt, const MarketShock& curveShock) : thisMarket(mkt)

{

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

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

curve.shock(curveShock.shock.first, curveShock.shock.second);

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

}

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

private:

Market thisMarket;

};

European Option and American Option DV01 can be computed similarly since they are also using Rate curve for discounting

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

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 tenor " << volShock.shock.first << "is shocked" << volShock.shock.second<<endl;

}

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

private:

Market thisMarket;

};

Trade creation - using Factory pattern

// Abstract creator class

class TradeFactory {

public:

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

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

}

};

// Concrete creator class - BondFactory

class BondFactory : public TradeFactory {

public:

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

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

}

};

// Concrete creator class - EuropeanFactory

class EurOptFactory : public TradeFactory {

public:

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

return std::make\_unique<EuropeanOption>(); //implement this

}

};

// Concrete creator class - AmericanOptFactory

class AmericanOptFactory : public TradeFactory {

public:

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

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

}

};

* it is ok to use share\_ptr as well
* avoid using raw pointer

Putting back everything togeher

int main()

{

// some sample code to demo

RiskEngine re;

string risk\_id = "usd-sofr:on";

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);

Market testMarketData;//original market

auto shockedUpCurveUp = CurveDecorator(testMarketData, testShockUp);

auto shockedUpCurveDown = CurveDecorator(testMarketData, testShockDown);

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

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

unordered\_map<string, double> thisDealDv01;

double pv\_up, pv\_down;

pv\_up = swap->Pv(shockedUpCurveUp.getMarket());

pv\_down = swap->Pv(shockedUpCurveDown.getMarket());

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

thisDealDv01.emplace(risk\_id, dv01);

}

We can define a class named RiskEngine try to encapsulate this computation.

class RiskEngine

{

RiskEngine(const Market& market, double shock) {

//add implementation

};

void 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: volSchoks) {

// 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 check\_even 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);

}

}

}

map<string, double> getResult();

private:

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

unordered\_map<string, VolDecorator> volSchoks;

map<string, double> result;

};