```cpp
#include "SampleExam3/SampleExam3.hpp"
#include "cfl/Interp.hpp"
#include "cfl/Macros.hpp"
#include <cmath>

using namespace cfl;
using namespace std;

cfl::Function prb::
discountSwapLogLinInterp(const std::vector<double> &rSwapRates,
                         double dPeriod, double dInitialTime)
{
  PRECONDITION(rSwapRates.size() > 0);

  //times for interpolation = initial time + payment times
  std::vector<double> uTimes(rSwapRates.size() + 1);
  uTimes.front() = dInitialTime;
  std::transform(uTimes.begin(), uTimes.end() - 1, uTimes.begin() + 1,
                 [dPeriod](double dX) { return dX + dPeriod; });

  std::vector<double> uLogDiscount(uTimes.size());
  uLogDiscount.front() = 0.;
  double dSum = 0;
  std::transform(rSwapRates.begin(), rSwapRates.end(), uLogDiscount.begin()+1,
                 [dPeriod, &dSum](double dRate)
                 {
                   double dD = (1.-dSum*dRate*dPeriod)/(1.+dRate*dPeriod);
                   dSum += dD;
                   return std::log(dD);
                 });

  //linear interpolation of the logs of discount factors
  cfl::Interp uLinear = NInterp::linear();
  Function uLogDiscountFunction =
    uLinear.interpolate(uTimes.begin(), uTimes.end(), uLogDiscount.begin());

  return cfl::exp(uLogDiscountFunction);
}
```

```cpp
#include "SampleExam3/SampleExam3.hpp"

using namespace cfl;
using namespace std;
using namespace prb;

namespace NForwardFXSvenssonFit
{
  cfl::Function yieldShape1(double dLambda, double dInitialTime)
  {
    std::function<double(double)> uShape =
      [dLambda, dInitialTime](double dT) {
        double dX = std::max(dLambda * (dT - dInitialTime), cfl::EPS);
        return (1 - std::exp(-dX)) / dX;
      };
    return Function(uShape, dInitialTime);
  }

  cfl::Function yieldShape2(double dLambda, double dInitialTime)
  {
    std::function<double(double)> uShape =
      [dLambda, dInitialTime](double dT) {
        double dX = std::max(dLambda * (dT - dInitialTime), cfl::EPS);
        return (1 - std::exp(-dX)) / dX - std::exp(-dX);
      };
    return Function(uShape, dInitialTime);
  }

  std::function<double(double, double)>
  forwardFX(double dSpotFX)
  {
    return [dSpotFX](double dDom, double dFor) {
             PRECONDITION(dDom > cfl::EPS);
             return dSpotFX * dFor / dDom;
           };
  }

  std::function<double(double, double)>
  costOfCarry(double dSpot, double dInitialTime)
  {
    return [dSpot, dInitialTime](double dF, double dT) {
             PRECONDITION(dT > dInitialTime + cfl::EPS);
             return std::log(dF / dSpot) / (dT - dInitialTime);
           };
  }

  cfl::Function
  forwardCarryFit(double dSpot,
                  const std::vector<double> &rTimes,
                  const std::vector<double> &rForwards,
                  double dInitialTime, cfl::Fit &rFit, cfl::Function &rErr)
  {
    //market cost-of-carry rates
    std::vector<double> uCostOfCarry(rTimes.size());
    std::transform(rForwards.begin(), rForwards.end(), rTimes.begin(),
                   uCostOfCarry.begin(), costOfCarry(dSpot, dInitialTime));

    rFit.assign(rTimes.begin(), rTimes.end(), uCostOfCarry.begin());
    Function uT([dInitialTime](double dT) { return dT - dInitialTime; }, dInitialTime);
    Function uForward = dSpot * exp(rFit.fit() * uT);
    rErr = uForward * uT * rFit.err();

    return uForward;
  }

  cfl::Function
  forwardFXCarryFit(double dSpotFX,
                    const std::vector<double> &rTimes,
                    const std::vector<double> &rDomesticDiscounts,
                    const std::vector<double> &rForeignDiscounts,
                    double dInitialTime, cfl::Fit &rFit, cfl::Function &rErr)
  {
```

```
    //forward exchange rates
    std::vector<double> uForwardFX(rTimes.size());
    std::transform(rDomesticDiscounts.begin(), rDomesticDiscounts.end(),
                   rForeignDiscounts.begin(), uForwardFX.begin(),
                   forwardFX(dSpotFX));

    return forwardCarryFit(dSpotFX, rTimes, uForwardFX,
                           dInitialTime, rFit, rErr);
  }
} // namespace NForwardFXSvenssonFit

using namespace NForwardFXSvenssonFit;

cfl::Function prb::
forwardFXSvenssonFit(double dSpotFX, const std::vector<double> &rTimes,
                     const std::vector<double> &rDomDF,
                     const std::vector<double> &rForDF,
                     double dLambda1, double dLambda2,
                     double dInitialTime, cfl::Function &rErr, FitParam &rParam)
{
  PRECONDITION(std::min(dLambda1, dLambda2) > 0);
  PRECONDITION(dLambda1 != dLambda2);

  std::vector<Function> uF = {Function(1.),
                              yieldShape1(dLambda1, dInitialTime),
                              yieldShape2(dLambda1, dInitialTime),
                              yieldShape2(dLambda2, dInitialTime)};
  Fit uFit = cfl::NFit::linear(uF);

  cfl::Function uForward = forwardFXCarryFit(dSpotFX, rTimes, rDomDF,
                                             rForDF, dInitialTime, uFit, rErr);
  rParam = uFit.param();
  return uForward;
}
```

```cpp
#include "SampleExam3/SampleExam3.hpp"

using namespace cfl;
using namespace std;

cfl::MultiFunction prb::
varianceSwapStrike(double dMaturity, unsigned iNumberOfTimes,
                   AssetModel & rModel)
{
  PRECONDITION(iNumberOfTimes>0);
  double dPeriod = (dMaturity - rModel.initialTime())/iNumberOfTimes;

  std::vector<double> uEventTimes(iNumberOfTimes+1, rModel.initialTime());
  std::transform(uEventTimes.begin(), uEventTimes.end()-1,
                 uEventTimes.begin()+1,
                 [dPeriod](double dX){ return dX+dPeriod; });
  rModel.assignEventTimes(uEventTimes);
  ASSERT(std::abs(rModel.eventTimes().back()-dMaturity)<= 10E-12);

  int iTime = rModel.eventTimes().size()-1;
  Slice uSum = rModel.cash(iTime, 0.);
  Slice uLogSpot = log(rModel.spot(iTime));
  Slice uValLogSpot = uLogSpot;

  while (iTime > 0) {
    //uSum is the value of the sum of squares of logs in the future
    //uLogSpot is the log of the current price of the stock
    //uValLogStop is the current value of log(uSpot) paid at maturity
    uSum += uLogSpot*uValLogSpot;
    iTime--;
    uSum.rollback(iTime);
    uValLogSpot.rollback(iTime);
    uLogSpot = log(rModel.spot(iTime));
    uSum -= 2*uLogSpot*uValLogSpot;
    uValLogSpot = uLogSpot*rModel.discount(iTime,dMaturity);
    uSum += uLogSpot*uValLogSpot;
  }
  ASSERT(iTime==0);

  uSum /= ((dMaturity-rModel.initialTime())*rModel.discount(0,dMaturity));
  Slice uStrike = sqrt(uSum);
  return interpolate(uStrike);
}
```

```cpp
#include "SampleExam3/SampleExam3.hpp"

using namespace cfl;
using namespace std;

cfl::MultiFunction prb::
callableCappedFloater(const Data::CashFlow & rCap, double dLiborSpread,
                      InterestRateModel & rModel)
{
  std::vector<double> uEventTimes(rCap.numberOfPayments);
  uEventTimes.front() = rModel.initialTime();
  std::transform(uEventTimes.begin(), uEventTimes.end()-1,
                 uEventTimes.begin()+1,
                 [&rCap](double dX){ return dX+rCap.period; });
  rModel.assignEventTimes(uEventTimes);

  //last minus one payment time
  int iTime = rModel.eventTimes().size() - 1;
  Slice uDiscount =
    rModel.discount(iTime, rModel.eventTimes()[iTime] + rCap.period);
  double dSpreadFactor = rCap.notional*(dLiborSpread*rCap.period - 1.);
  double dCapFactor = rCap.notional*rCap.rate*rCap.period;
  Slice uValueOfNextCoupon =
    min(rCap.notional + dSpreadFactor*uDiscount, dCapFactor*uDiscount);
  //uOption is the value to continue (includes next coupon)
  Slice uOption = rCap.notional*uDiscount + uValueOfNextCoupon;
  while (iTime >0) {
    //uOption is the value to continue (includes the value of the next coupon).
    uOption = min(uOption, rCap.notional);
    iTime--;
    uOption.rollback(iTime);
    uDiscount =
      rModel.discount(iTime, rModel.eventTimes()[iTime] + rCap.period);
    uValueOfNextCoupon =
      min(rCap.notional + dSpreadFactor*uDiscount, dCapFactor*uDiscount);
    uOption += uValueOfNextCoupon;
  }
  return interpolate(uOption);
}
```