```cpp
#include "Homework2/Homework2.hpp"
#include "cfl/Interp.hpp"

using namespace cfl;
using namespace std;

namespace NForwFXCarryInterp
{

  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);
    };
  }
} // namespace NForwFXCarryInterp

using namespace NForwFXCarryInterp;

cfl::Function prb::
    forwardFXCarryLinInterp(double dSpotFX,
                            const std::vector<double> &rTimes,
                            const std::vector<double> &rDom,
                            const std::vector<double> &rFor,
                            double dInitialTime)
{
  PRECONDITION(rTimes.size() == rDom.size());
  PRECONDITION(rTimes.size() == rFor.size());
  PRECONDITION(rTimes.size() > 0);
  PRECONDITION(rTimes.front() > dInitialTime);
  PRECONDITION(std::is_sorted(rTimes.begin(), rTimes.end(),
                              std::less_equal<double>()));

  //times for interpolation = initial time + discount maturities
  std::vector<double> uTimes(rTimes.size() + 1);
  uTimes.front() = dInitialTime;
  copy(rTimes.begin(), rTimes.end(), uTimes.begin() + 1);

  //market forward exchange rates
  std::vector<double> uForwardFX(rTimes.size());
  std::transform(rDom.begin(), rDom.end(),
                 rFor.begin(), uForwardFX.begin(),
                 forwardFX(dSpotFX));
  //market cost-of-carry rates
  std::vector<double> uCarry(uTimes.size());
  std::transform(uForwardFX.begin(), uForwardFX.end(), rTimes.begin(),
                 uCarry.begin() + 1, costOfCarry(dSpotFX, dInitialTime));

  //cost-of-carry rate is constant on the first interval
  uCarry.front() = uCarry[1];

  //linear interpolation of the cost-of-carry rates
  cfl::Interp uLinear = NInterp::linear();
  Function uCarryFunction =
      uLinear.interpolate(uTimes.begin(), uTimes.end(), uCarry.begin());

  return cfl::Data::forward(dSpotFX, uCarryFunction, dInitialTime);
}
```

```cpp
#include "Homework2/Homework2.hpp"
#include "cfl/Interp.hpp"
#include "cfl/Macros.hpp"
#include <cmath>

using namespace cfl;
using namespace std;

cfl::Function prb::
discountLogInterp(const std::vector<double> &rTimes,
                  const std::vector<double> &rDiscount,
                  double dInitialTime, const cfl::Interp &rInterp)
{
  PRECONDITION(rTimes.size() == rDiscount.size());
  PRECONDITION(rTimes.size() > 0);
  PRECONDITION(rTimes.front() > dInitialTime);
  PRECONDITION(std::is_sorted(rTimes.begin(), rTimes.end(),
                              std::less_equal<double>()));

  //times for interpolation = initial time + discount times
  std::vector<double> uTimes(rTimes.size()+1);
  uTimes.front() = dInitialTime;
  std::copy(rTimes.begin(), rTimes.end(), uTimes.begin()+1);

  //logs of discount factors
  std::vector<double> uLogDiscount(uTimes.size());
  uLogDiscount.front() = 0.;
  std::transform(rDiscount.begin(), rDiscount.end(), uLogDiscount.begin() + 1,
                 [](double dX) { return std::log(dX); });

  //interpolation of the logs of discount factors
  Function uLogDiscountFunction =
    rInterp.interpolate(uTimes.begin(), uTimes.end(), uLogDiscount.begin());

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

```cpp
#include <numeric>
#include "Homework2/Homework2.hpp"
#include "cfl/Data.hpp"
#include "cfl/Macros.hpp"

using namespace cfl;
using namespace std;
using namespace prb;
using namespace cfl::Data;

namespace NVolBlackFit
{
  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);
  }
} // namespace NVolBlackFit

using namespace NVolBlackFit;

cfl::Function prb::
    volatilityBlackFit(const std::vector<double> &rTimes,
                       const std::vector<double> &rVols,
                       double dLambda, double dInitialTime,
                       Function &rErr, FitParam &rParam)
{
  PRECONDITION(rTimes.size() == rVols.size());
  PRECONDITION(rTimes.size() > 0);
  PRECONDITION(rTimes.front() > dInitialTime);

  Function uBase = sqrt(yieldShape1(2 * dLambda, dInitialTime));
  Fit uFit = NFit::linear(uBase);
  uFit.assign(rTimes.begin(), rTimes.end(), rVols.begin());
  rParam = uFit.param();
  rErr = uFit.err();
  return uFit.fit();
}
```

```cpp
#include "Homework2/Homework2.hpp"

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

namespace NSvenssonFit
{
  cfl::Function
  discountYieldFit(const std::vector<double> &rTime, const std::vector<double> &rDF,
                   double dInitialTime, cfl::Fit &rFit, Function &rErr)
  {
    PRECONDITION(rTime.size() == rDF.size());
    PRECONDITION(rTime.size() > 0);
    PRECONDITION(rTime.front() > dInitialTime);
    PRECONDITION(std::is_sorted(rTime.begin(), rTime.end(),
                                std::less_equal<double>()));

    std::vector<double> uYield(rTime.size());
    std::transform(rTime.begin(), rTime.end(), rDF.begin(), uYield.begin(),
                   [dInitialTime](double dT, double dD) {
                     ASSERT(dT - dInitialTime > cfl::EPS);
                     return -std::log(dD) / (dT - dInitialTime);
                   });
    rFit.assign(rTime.begin(), rTime.end(), uYield.begin());

    Function uT([dInitialTime](double dT) { return dT - dInitialTime; },
                dInitialTime);
    Function uDF = exp(-rFit.fit() * uT);

    rErr = rFit.err() * uDF * uT;

    return uDF;
  }

  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);
  }
} // namespace NSvenssonFit

using namespace NSvenssonFit;

cfl::Function prb::
    discountSvenssonFit(const std::vector<double> &rTimes, const std::vector<double> &rDF,
                        double dLambda1, double dLambda2, double dInitialTime,
                        Function &rErr, FitParam &rParam)
{
  PRECONDITION(std::min(dLambda1, dLambda2) > 0);
  PRECONDITION(dLambda1 != dLambda2);
  PRECONDITION(rTimes.front() > dInitialTime);
  PRECONDITION(is_sorted(rTimes.begin(), rTimes.end(), std::less_equal<double>()));

  std::vector<Function> uF = {Function(1.),
                              yieldShape1(dLambda1, dInitialTime),
                              yieldShape2(dLambda1, dInitialTime),
                              yieldShape2(dLambda2, dInitialTime)};
```

```
  Fit uFit = cfl::NFit::linear(uF);
  Function uD = discountYieldFit(rTimes, rDF, dInitialTime, uFit, rErr);
  rParam = uFit.param();
  return uD;
}
```