LECTURE 11

Regression

```
1  template<typename PPI, typename F>
2  class regression
3  {
4  public:
5    typedef PPI path_piece_in;//!< input noise components
6    typedef boost::numeric::ublas::matrix<mc::scalar> bmatrix;//!< bost
       matrix of scalars
7    typedef F Function;//!< type of coefficient function
8    typedef std::vector<Function> fvector; //!< vector of dop::Function's
9    typedef typename mc::mc_factory<path_piece_in,mc::bvector> factory;//!<
       particular factory
10   typedef typename factory::path_in_ptr path_in_ptr;//!<pointer to input
       noise
11   typedef typename factory::path_out path_out;//!<solution path
12   typedef mc::bvector path_piece_out;//!<solution path components
13   virtual ~regression(){}
14   virtual mc::bvector & operator()(const path_piece_out & factorAt0,
15                mc::bvector & bvOut)=0;
16 protected:
17   bmatrix Regression(const std::vector<mc::bvector> & yVals,
18            const std::vector<path_piece_out> & factors,
19            const fvector & testFunctions);
20   mc::bvector & Projection(const path_piece_out & factor,
21             const fvector & testFunctions,
22             const bmatrix & coefficients,
23             mc::bvector & bvOut);
24
25 };
```

Least squares based pricing methods

- gives estimate in form of linear combination of test functions
- the coefficients are computed through (a series of) least squares regression(s)

`regression`: abstract class for least squares based pricing methods
template arguments

- `PPI`: "path piece in", solution path assumed to be built of vectors
- `F`: type of test functions - not specified in `MClib`, in the main `DOPlib` is used

Common properties

- `Regression`: computes least squares regression, `protected` member
- `Projection`: given the regression coefficients, computes projection, `protected` member
- `operator()`: the actual pricing function

Note: `protected`: members can be directly accessed by classes derived from `regression`.

Bermudan options

Dynamic programming principle:

$$V(t_n, S_{t_n}) = \max\left\{f(S, t_n)), D_{t_n,t_{n+1}}\mathbb{E}\left[V(t_{n+1}, S_{t_{n+1}})|S_{t_n}\right]\right\}$$

We use backward recursion and least squares regression to estimate $V(0, S)$. In particular,

- start with a grid of initial values $(S_0^{(1)}, \ldots, S_0^{(M)})$
- simulate trajectories: $S_0^{(i)}, S_{t_1}^{(i)}, \ldots, S_{t_N}^{(i)}$ for $i = 1, \ldots, M$
- set $V(t_N, S) = f(S, t_N)$
- then for $n = N-1, \ldots, 1$, recursively apply the dynamic programming principle with

$$\mathbb{E}\left[V(t_{n+1}, S_{t_{n+1}})|S_{t_n} = S\right] \approx \sum_r \phi_r(S)\beta_{n,r}$$

where the regression coefficients are estimated given $S_{t_n}^{(1)}, \ldots, S_{t_n}^{(M)}$ and $V(t_{n+1}, S_{t_{n+1}}^{(1)}), \ldots, V(t_{n+1}, S_{t_{n+1}}^{(M)})$

- once computed $\beta_{0,r}$ for $r = 1, \ldots, R$, save them.

Implementation

To model $f(S,t)$ (actually, to model $f(S,[s,t])$), we use time dependent payoffs

```
template<typename path_piece_out, class PM
= typename mc::path_piece_types<path_piece_out>::path_map_by_right_end
>
class time_dependent_payoff
{
public:
  typedef mc::path<path_piece_out,PM> path_out;//!<solution path
  virtual ~time_dependent_payoff(){}
  virtual mc::bvector & operator()(path_out & pFactors,
                const mc::dyadic & dTimeStep,
                mc::bvector & bvValue)=0;
  virtual unsigned int SizePayoff() const=0;
};
```

`operator()` is its key member

`bermudan_option_pricer` is derived from `regression`. Its constructor:

```
template<typename PPI, typename F>
pmc::bermudan_option_pricer<PPI,F>
::bermudan_option_pricer(const std::vector<path_piece_out> & factorsAt0,
          unsigned int iNumericalSDEAccuracy,
          unsigned int iBermudanScaleAccuracy,
          mc::scalar sT,
          mc::scalar sR,
          factory & ParticularFactory,
          time_dependent_payoff & ParticularTimedependentPayoff,
          const fvector & testFunctions)
  : m_TestFunctions(testFunctions)
```

Arguments

- grid of initial values
- accuracy arguments: numerical SDE scale, Bermudan exercise times
- interest rate, time scaling factor
- factory, time dependent payoff
- collection of test functions

```
{
  typename std::vector<path_piece_out>::size_type n(factorsAt0.size());
  std::vector<path_out> trajectories;
  trajectories.reserve(n);
  std::vector<mc::bvector> yVals(n);
  mc::dyadic dyTime((1<<iBermudanScaleAccuracy)-1,iBermudanScaleAccuracy)
       ;
  for(unsigned int i=0; i<n; ++i){
    trajectories.push_back(ParticularFactory.GetNewPathOut(
         iNumericalSDEAccuracy,
                        iBermudanScaleAccuracy,
                        factorsAt0[i],
                        sT));
    ParticularTimedependentPayoff(trajectories.back(),dyTime,yVals[i]) *=
      std::exp(-sR*sT);
  }
```

- this loop generates trajectories
- and evaluates payoff at maturity

```
    //backward induction
  std::vector<path_piece_out> factorsAtT(n);
  mc::bvector tempExercise(ParticularTimedependentPayoff.SizePayoff());
  mc::bvector tempContinuation(ParticularTimedependentPayoff.SizePayoff()
       );
  std::pointer_to_binary_function <const mc::scalar &,
                const mc::scalar &,
                const mc::scalar &> maxOp(std::max<mc::scalar>);
  --dyTime;
```

- preparation of backward induction
- `maxOp` is a function object created by a STL template, ref `http://www.cplusplus.com/reference/std/functional/pointer_to_binary_function/`

```
for(int j=0; j < (1<<iBermudanScaleAccuracy)-1 ; ++j, --dyTime ){
    // accessing factor values at time dyTime
  for(unsigned int i=0; i<n; ++i)
    factorsAtT[i]=trajectories[i].find(dyTime)->second;

    // regression for continuation value
  m_Coefficients=this->Regression(yVals,factorsAtT,m_TestFunctions);

  // dynamic programming principle for value
 // working out new values comparing continuation and exercise values
  for(unsigned int i=0; i<n; ++i){
    // continuation value
    this->Projection(factorsAtT[i],m_TestFunctions,m_Coefficients,
        tempContinuation);
    // exercise value
    mc::scalar sTTemp=dyTime.GetLength()*mc::scalar(dyTime.Position()+1)
        *sT;

    ParticularTimedependentPayoff(trajectories[i],dyTime,tempExercise)
        *=
  std::exp(-sR*sTTemp);
    //new values
    std::transform(tempExercise.begin(),tempExercise.end(),
        tempContinuation.begin(),yVals[i].begin(),maxOp);
  }
}
```

Backward induction: for time t_n (previous slide):

- `yVals` stores latest values $V(t_{n+1}, S_{t_{n+1}})$
- first inner-loop writes factor values S_{t_n} into `factorsAtT`
- `Regression()` is called with `yVals` and `factorsAtT`
- second inner loop applies the dynamic programming principle on each trajectory using the time dependent payoff and `Projection()`
- recall that `dyadic` models subintervals of $[0, 1]$ of the form $[k2^{-n}, (k+1)2^{-n}]$
- useful members of `dyadic`: `Position()` returns k, `Accuracy()` returns n, `GetLength()` returns 2^{-n}

Last step

```
  // last step - regression at time 0 (no payoff assumed at initial time)
  m_Coefficients=this->Regression(yVals,factorsAt0,m_TestFunctions);
}
```

- The conditional expectation of the t_1 values are estimated

operator()

```
1  template<typename PPI, typename F>
2  mc::bvector & pmc::bermudan_option_pricer<PPI,F>
3  ::operator()(const path_piece_out & factorAt0, mc::bvector & bvOut)
4  {
5    return this->Projection(factorAt0,m_TestFunctions,m_Coefficients,bvOut)
         ;
6  }
```

- operator() is based on projection using the data members
- m_TestFunctions
- m_Coefficients
- note that operator() takes bvOut by reference and returns a reference to it

An example

```
1     //setting up payoff
2     mc::bvector bvStrikes(3);
3     bvStrikes[0]=0.8;
4     bvStrikes[1]=1.0;
5     bvStrikes[2]=1.2;
6     pmc::time_dependent_put putPayoff(bvStrikes);
7     //setting up grid of initial values
8     unsigned int iNumberOfGridPoints=10000;
9     //...
10    //setting up test functions
11    bermudan_option_pricer::fvector testFunctions;
12    //...
13    //initialising the option pricer
14    bermudan_option_pricer europeanPutPricer(factorsAt0, iAccuracy,0, sT,
          sR,
15                    bsFactory, putPayoff, testFunctions);
16    //setting up test values for initial condition
17    std::vector<mc::bvector> testVals;
18    //...
19       bermudanPutPricer(testVals[i],bvEstimates);
```

Description of the example code

- the particular time dependent payoff is `time_dependent_put`
- initial grid of factors
- test functions - `DOPlib` is used
- an instance of `bermudan_option_pricer` is constructed
- a set of test values is constructed
- `operator()` of the pricer is called

The output

```
*********************************************
*          BERMUDAN OPTION PRICER           *
*********************************************
Parameters of put option:
Risk-free rate: 0.05
Volatility: 0.2
Time to maturity: 0.5
Strikes: [3](0.8,1,1.2)
Stock price: 0.78
   Estimated bermudan option prices: 0.0518558, 0.218213, 0.4184
   Estimated european option prices: 0.0434198, 0.196693, 0.3890
   Values by Black-Scholes formula:  0.0441012, 0.198308, 0.3904
Stock price: 0.8
   Estimated bermudan option prices: 0.0429653, 0.198257, 0.3982
   Estimated european option prices: 0.0351983, 0.178821, 0.3700
   Values by Black-Scholes formula:  0.0353578, 0.179871, 0.3705
Stock price: 0.82
   Estimated bermudan option prices: 0.0353555, 0.179021, 0.3780
   Estimated european option prices: 0.028271, 0.161561, 0.35105
   Values by Black-Scholes formula:  0.0279794, 0.162017, 0.3506
Stock price: 0.84
   Estimated bermudan option prices: 0.02889, 0.16067, 0.35793
   Estimated european option prices: 0.0224798, 0.145002, 0.3319
   Values by Black-Scholes formula:  0.021858, 0.144868, 0.33082
```