LECTURE 12

Payoff

Source code: `MCLib/base_payoff_statistics.hpp`

```
template<class path_piece_out, class PM
>
class payoff
{
public:
  typedef typename mc::path<path_piece_out, PM> path_out;//!<solution
       path
  virtual ~payoff(){}
  virtual mc::bvector & operator()(path_out & poArg,
                mc::bvector & bvOut)=0;
  virtual unsigned int SizePayoff() const=0;
};
```

- `payoff` is an abstract class defining a framework for payoffs of the type $f(S.)$
- high-level functions take reference or pointer to `payoff`
- `virtual` members to be implemented in derived classes: `operator()`, `SizePayoff()`
- `operator()` take a solution path and a boost vector `bvOut` by reference and must return a reference to `bvOut`
- `SizePayoff()` returns the size of the payoff vector, can be 0 is the size is unknown before the call of `operator()` (this is subject to revision)

European call - declaration

Source code: `MCLib/particular_payoff_statistics.hpp`

```
class european_calls_payoff :
   public mc::payoff<mc::bvector>
{
public:
  european_calls_payoff(const mc::bvector & bvStrikes,
          unsigned int ind=0);
  mc::bvector & operator()(path_out & poArg,
            mc::bvector & bvOut);
  unsigned int SizePayoff() const;

private:
  mc::bvector bvStrikes_;//!< vector of strikes
  unsigned int ind_;//!< index of the underlying in path_piece for option
       payoff
};
```

- `european_calls_payoff` is a particular class derived from `payoff`
- it stores a vector of strikes $K = (K_1, \ldots, K_n)$ in `bvStrikes_` and an index i in `ind_`
- `operator()` returns a vector of call payoff values: $P = (P_1, \ldots, P_n)$, where

$$P_k = \max(S_T[i] - K_k, 0), \text{ for } k = 1, \ldots, n$$

and S_T is the path piece of the solution path corresponding to time T

European call - implementation

Source code: `MCLib/particular_payoff_statistics.cpp`

```
1  pmc::european_calls_payoff::
2  european_calls_payoff(const mc::bvector & bvStrikes,
3              unsigned int ind)
4    : bvStrikes_(bvStrikes),
5      ind_(ind)
6  {}
```

- the constructor initialises the data members

```
1  unsigned int pmc::european_calls_payoff::SizePayoff() const {
2    return bvStrikes_.size();
3  }
```

- `SizePayoff()` returns the size of vector of strikes

```
1  mc::bvector & pmc::european_calls_payoff::
2  operator()(path_out & poArg, mc::bvector & bvOut)
3  {
4    const mc::bvector & underlying=poArg.find(mc::dyadic(0,0))->second;
5    size_t n=SizePayoff();
6    bvOut.resize(n);
7
8    for(size_t i=0; i<n; ++i)
9      bvOut[i]=std::max(underlying[ind_]-bvStrikes_[i],0.0);
10
11   return bvOut;
12 }
```

- S_T is accessed using the `find()` member of `poArg` with the dyadic interval $[0, 1]$
- `bvOut` is resized to the right size
- the loop evaluates each of the call payoffs
- a reference to `bvOut` is returned

Source code: MCLib/particular_payoff_statistics.hpp

```
class asian_discretely_sampled_payoff :
   public mc::payoff<mc::bvector>
{
public:
  asian_discretely_sampled_payoff(unsigned int iSamplingAccuracy,
              unsigned int indY=0);
  mc::bvector & operator()(path_out & poArg,
            mc::bvector & bvOut);
  unsigned int SizePayoff() const;
private:
  unsigned int iSamplingAccuracy_;//!< the average is taken over a scale
      of 2^{iSamplingAccuracy) steps
  unsigned int indY_;//!< the index of the component to be averaged
};
```

- `asian_discretely_sampled_payoff` is another particular class derived from `payoff`
- Discretely sampled Asian call pays

$$\max\left(S_T[i] - \frac{1}{M}\sum_{k=1}^{M} S_{Tk2-M}[i], 0\right)$$

- `iSamplingAccuracy_` stores M
- `indY_` stores the index i

Source code: MCLib/particular_payoff_statistics.cpp

```
pmc::asian_discretely_sampled_payoff::
asian_discretely_sampled_payoff(unsigned int iSamplingAccuracy,
            unsigned int indY)
  : iSamplingAccuracy_(iSamplingAccuracy),
    indY_(indY)
{}
```

- the constructor initialises the data members

```
unsigned int pmc::asian_discretely_sampled_payoff::SizePayoff() const
{
  return 1;
}
```

- `SizePayoff()` returns 1

```
mc::bvector & pmc::asian_discretely_sampled_payoff::
operator()(path_out & poArg, mc::bvector & bvOut)
{
  bvOut.resize(SizePayoff());
  mc::dyadic dyadStep(0,iSamplingAccuracy_);
  unsigned int n=(1<<iSamplingAccuracy_)-1;
  mc::scalar AvY(mc::scalar(0));
  mc::scalar multiplier;
  for(unsigned int i=0; i<n; ++i, ++dyadStep){
    multiplier=mc::scalar(i)/mc::scalar(i+1);
    AvY*=multiplier;
    AvY+=((*(poArg.find(dyadStep))).second)[indY_]/mc::scalar(i+1);
  }
  bvOut[0]=std::max(((*(poArg.find(dyadStep))).second)[indY_]-AvY,
          mc::scalar(0));
  return bvOut;
}
```

`operator()`

- `bvOut` is resized
- a compatible `iSamplingAccuracy` is computed
- the `dyadic` $[0, 2^{-M}]$ is initialised
- `n`=2^M using the bit shift operator << (note that it is not the `ostream` operator because the first argument is not an `ostream`)
- in the loop the average stock price is computed, note that both `i` and the dyadic interval `dyadStep` are incremented in each cycle
- the payoff value is written into `bvOut`
- a reference to `bvOut` is returned

Further classes derived from `payoff` declared in `particular_payoff_statistics.hpp`:

- `asian_continuously_sampled_payoff`
- `lookback_discretely_sampled_payoff`

and another one declared in `particular_ko_payoff.hpp`:

- `knock_out_payoff`

Source code: `MCLib/particular_ko_payoff.hpp`

```
template<typename ko_cond>
class knock_out_payoff : public mc::payoff<mc::bvector>
{
public:
  typedef sptr::shared_ptr<mc::payoff<mc::bvector> > payoff_ptr;//!<smart
       pointer to payoff
  knock_out_payoff(payoff_ptr PtrPayoff,
         ko_cond KOCond,
         unsigned int iPathDepAccuracy=0);
  knock_out_payoff(mc::payoff<mc::bvector> * PtrPayoff,
         ko_cond KOCond,
         unsigned int iPathDepAccuracy=0);
  mc::bvector & operator()(path_out & poArg, mc::bvector & bvOut);
  unsigned int SizePayoff() const;

private:
  payoff_ptr PtrPayoffIfNotKO_;//!< points to the payoff component
  ko_cond KOCond_;//!< template type function object, bool operator(
       bvector) member is assumed
  unsigned int iPathDepAccuracy_;//!< defines the scale on which the KO
       kondition is checked
};
```

General knock out payoffs - declaration

- `knock_out_payoff` is a particular class derived from `payoff`
- `knock_out_payoff` implements payoffs of the type

$$P(S)\left(1-\mathbf{1}_{A(S_t,t\in\{k2^{-M}T|k=1,\ldots,2^{-M}\})}\right)$$

  where $P(\cdot)$ is any payoff and $A(\cdot)$ is a knock-out event, i.e. if A does not occur, the option pays P, otherwise it pays 0
- the member `PtrPayoffIfNotKO_` is the pointer to the payoff $P(\cdot)$
- `KOCond_` is of type `ko_cond` (the template type), implements $\mathbf{1}_{A(\cdot)}$
- `iPathDepAccuracy_` defines the sampling accuracy, i.e. M

Implicit assumption on particular `ko_cond` objects - must have these members

```
  bool operator()(const mc::bvector & bvArg) const;
  void Reset(){}
```

- `operator()` takes a boost vector (particular solution path piece) and returns a `bool`
- `Reset()` might be used if the state of the `ko_cond` is subject to changes

General knock out payoffs - implementation

```
template<typename ko_cond>
mc::bvector & pmc::knock_out_payoff<ko_cond>::
operator()(path_out & poArg, mc::bvector & bvOut)
{
  unsigned int iPathDepAccuracy=
    std::min(iPathDepAccuracy_,poArg.MaxAccuracy());
  KOCond_.Reset();
  bool isNotKO(true);
  mc::dyadic dyadStep(0,iPathDepAccuracy);
  unsigned int i=0;
  unsigned int iMaxSteps=(1<<iPathDepAccuracy);
  while(isNotKO && i<iMaxSteps){
    isNotKO= !(KOCond_(poArg.find(dyadStep)->second));
    ++dyadStep;
    ++i;
  }
  if(isNotKO)
    return ((*PtrPayoffIfNotKO_)(poArg,bvOut));
  else {
    bvOut.resize(SizePayoff());
    return (bvOut*=mc::scalar(0));
  }
}
```

`operator()` implementation (previous slide)

- computing a a compatible sampling accuracy
- reset KO condition
- defining loop variables
- `while` loops until KO condition is met or until T is reached
- if not knocked out, evaluate payoff
- if knocked out, return zero vector

```
class UpAndOut
{
public:
  UpAndOut(mc::scalar sBarrier,
      unsigned int ind=0);
  bool operator()(const mc::bvector & bvArg) const;
  void Reset(){}
private:
  mc::scalar sBarrier_;//!< KO barrier
  unsigned int ind_;//!< index of component triggering the KO
};
```

- required members: `operator()` and `Reset()`
- additional members: `sBarrier_` barrier B, `ind_` index i of component triggering the knock out event
- knock out event occurs at time t if $Y_t[i] \geq B$

Further knock out conditions implemented in `MClib` declared in `particular_ko_conditions.hpp`

- `DoubleBarrier`
- `UpDownAndOut`

```
pmc::UpAndOut::UpAndOut(mc::scalar sBarrier,
          unsigned int ind)
  : sBarrier_(sBarrier), ind_(ind)
{}
```

- the constructor initialises the data members

```
bool pmc::UpAndOut::operator()(const mc::bvector & bvArg) const
{
  return bvArg[ind_]>=sBarrier_;
}
```

- `operator()` returns `true` is $Y_t[i] \geq B$

Source code: `MCLib/base_payoff_statistics.hpp}`

```
template<typename path_piece_out, class PM
>
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;
};
```

- `time_dependent_payoff` is an abstract class, defining the framework for payoffs of the type $f(S., [s,t])$ where $[s,t]$ is a dyadic interval
- high-level functions take reference or pointer to `time_dependent_payoff`
- the concept is similar to `payoff`
- the difference is in `operator()` that also takes a `dyadic` as argument

Source code: `MCLib/particular_payoff_statistics.hpp`

```
class time_dependent_put :
   public mc::time_dependent_payoff<mc::bvector>
{
public:
  time_dependent_put(const mc::bvector & bvArg,
           mc::bvector::size_type index=0);
  mc::bvector & operator()(path_out & pFactors,
            const mc::dyadic & dTimeStep,
            mc::bvector & bvValue);
  unsigned int SizePayoff() const;
private:
  mc::bvector::size_type m_index;//!< index of the underlying in
       path_piece for option payoff
  mc::bvector m_bvStrikes;//!< vector of strikes
};
```

- `time_dependent_put` is a particular class derived from `time_dependent_payoff`
- this function object implements the time dependent payoff

$$f(S_\cdot, [s,t]) = \max(K_i - S_t, 0)$$

for a vector of strikes $K = (K_1, \ldots, K_n)$

Source code: `MCLib/particular_payoff_statistics.cpp`

```
mc::bvector & pmc::time_dependent_put::
operator()(path_out & pFactors,
      const mc::dyadic & dTimeStep,
      mc::bvector & bvValue)
{
  const mc::bvector & underlying=pFactors.find(dTimeStep)->second;
  size_t n=SizePayoff();
  bvValue.resize(n);
  for(size_t i=0; i<n; ++i)
    bvValue[i]=std::max(m_bvStrikes[i]-underlying[m_index],0.0);

  return bvValue;
}
```

- `operator()` looks up S_t in S.
- then computes the put payoff with each of the strikes, results are written into `bvValue`
- the function returns reference to `bvValue`

Summary

Key functions

- `path<path_piece>::find(dyadic dyad)` returns an iterator to the path piece corresponding to `dyad`
- `dyadic::dyadic(unsigned int k, unsigned int n)` - constructs $[k2^{-n}, (k+1)2^{-n}]$
- `dyadic::operator++()` - increments k (does not leave $[0, 1]$)
- `dyadic::operator--()` - decrements k (does not leave $[0, 1]$)
- `dyadic::Accuracy()` returns n
- `dyadic::Position()` returns k
- `dyadic::GetLength()` returns 2^{-n}
- `1<<n` - bit shift operator

Closing thoughts

Bazinga!

Sheldon Cooper