LECTURE 10

The high level interface, an example

```
1  typedef mc::mc_factory<PPI,PPO> factory;
2  mc::bvector bvStocks;
3  for(unsigned int i=0; i<iNumberOfPaths; ++i)
4  {
5   path_out poPath(ParticularFactory.GetNewPathOut(iAccuracyUsed,
        iGlobalAccuracy,ppoInCond,sT));
6   ParticularPayoff(poPath,bvStocks);
7   ParticularStatistics.DumpOnePayoff(bvStocks);
8  }
```

- The factory `myFactory` is initialized with a certain SDE determined by the object `esSetup`
- the member `GetNewPathOut()` of `myFactor` **generates a new price trajectory** (the numerical method takes $2^{\texttt{iAccuracyLimit},}$ steps, however the trajectory is saved on a scale with $2^{\texttt{iGlobalAccuracy}}$ pieces, the initial condition is set by `ppoInCond`, and the time horizon by `sT`)
- The **payoff** is computed by the function object `ParticularPayoff`, the object `ParticularStatistics` keeps track of the simulation **statistics**.

The high level interface, an example - continued


```
1  typedef mc::mc_factory<PPI,PPO> factory;
2  mc::bvector bvStocks;
3  for(unsigned int i=0; i<iNumberOfPaths; ++i)
4  {
5   path_out poPath(ParticularFactory.GetNewPathOut(iAccuracyUsed,
        iGlobalAccuracy,ppoInCond,sT));
6   ParticularPayoff(poPath,bvStocks);
7   ParticularStatistics.DumpOnePayoff(bvStocks);
8  }
```


- The main code, containing these lines, is based on: Brownian input noise (increments), equal step-size global numerical SDE strategy, Euler local steps, and the geometric Brownian motion SDE.
- To write high level code (e.g. the body of the `for` loop) does not require the detailed knowledge of the components described in the previous point. **For various combinations of components, the same high level code is applicable.**
- MC simulation based pricing algorithm of many financial products can be implemented only by writing a proper payoff function object, again independently from the components in the background.
- On the other hand, one can implement fancy **input noise models**, **input noise generators**, **SDE solvers** and other components together with a **factory** assembling a matching and working composition. The above high level code will run with this new factory too.

The high level interface - `mc_factory`

Description of the interface of `mc_factory<PPI,PPO>`

```
1  virtual path_out GetNewPathOut(unsigned int iAccuracyLimit,
2              unsigned int iGlobalAccuracy,
3              path_piece_out ppoArg,
4              mc::scalar sT)=0;
```

Returns an output path (a trajectory of the solution to the SDE).

- `iAccuracyLimit` sets the scale for the local steps (e.g. log2-number of Euler steps)
- `iGlobalAccuracy` sets the log2 scale of the solution (=0 for European options)
- `ppoArg` defines the initial condition.
- `sT` defines the time horizon

The high level interface - `mc_factory`

Description of the interface of `mc_factory<PPI,PPO>`

```
1  virtual path_out GetNewPathOut(path_in_ptr pipArg,
2              unsigned int iAccuracyLimit,
3              unsigned int iGlobalAccuracy,
4              path_piece_out ppoArg,
5              mc::scalar sT)=0;
```

Similar to the previous version, however this one takes a `path_in_ptr` (smart pointer) to an input path trajectory (which has at least $2^{\text{iAccuracyLimit}}$ many steps).

```
1  virtual path_in_ptr GetNewPathInPtr(unsigned int iAccuracyLimit,
2              mc::scalar sT)=0;
```

Returns a pointer to a newly generated input noise trajectory with (at least) $2^{\text{iAccuracyLimit}}$ many steps. `sT` defines the time horizon.

The high level interface - payoff and statistics

payoff<path_piece_out>

```
payoff<path_piece_out> (abstract)
-----------------------------------------------------------------
mc::bvector & operator()(path_out & poArg, bvector & bvOut)
unsigned int SizePayoff() const
```

- `payoff` is an abstract function object, its operator `()` takes a solution path and returns a (reference to the input) vector of payoff values (multiple products can be priced along the same solution path simultaneously).
- `SizePayoff()` returns the size of the payoff vector to be computed.
- MClib provides two particular `payoff<bvector>`'s, namely: `european_calls_payoff` and `knock_out_payoff`.

The high level interface - payoff and statistics

statistics

```
statistics (abstract)
-----------------------------------------------------------------
void DumpOnePayoff(bvector & bvArg)
std::vector<mc::bvector> GetStatistics()
```

- Its `DumpOnePayoff()` takes a `bvector` filled with payoff values (e.g. by a particular `payoff`) and updates its record of statistics.
- The member `GetStatistics()` returns a container of `bvector`'s, each containing some statistics (e.g. the first contains the means of the estimates, the second the estimated variances, etc.)
- MClib provides a particular `statistics`: `mean_variance_statistics`.

Note the separation of responsibilities.

Types defined or used by the MClib

The following types are defined in the MClib

- `dyadic` - model of sub-intervals of $[0, 1]$ of the form $[k2^{-n}, (k+1)2^{-n}]$.
- `path<path_piece>` - model of paths, behaves like a `map<dyadic,path_piece>` with some restrictions (see later)
- `path_generator<path_piece>` - abstract class, general model of path generators, component of `path<path_piece>`
- `solution_path_generator<PPI,PPO>` - a particular path generator based on numerical approximations of SDEs
- `strategy<PPI,PPO>` - abstract class, component of `solution_path_generator<PPI,PPO>`, defines the interface of global numerical SDE strategies
- `numerical_step<PPI,PPO>` - abstract class, component of `solution_path_generator<PPI,PPO>`, defines the interface of local numerical steps
- `payoff<PO>` - abstract class, defines the interface of payoff functions
- `statistics` - abstract class, defines the interface of payoff statistics

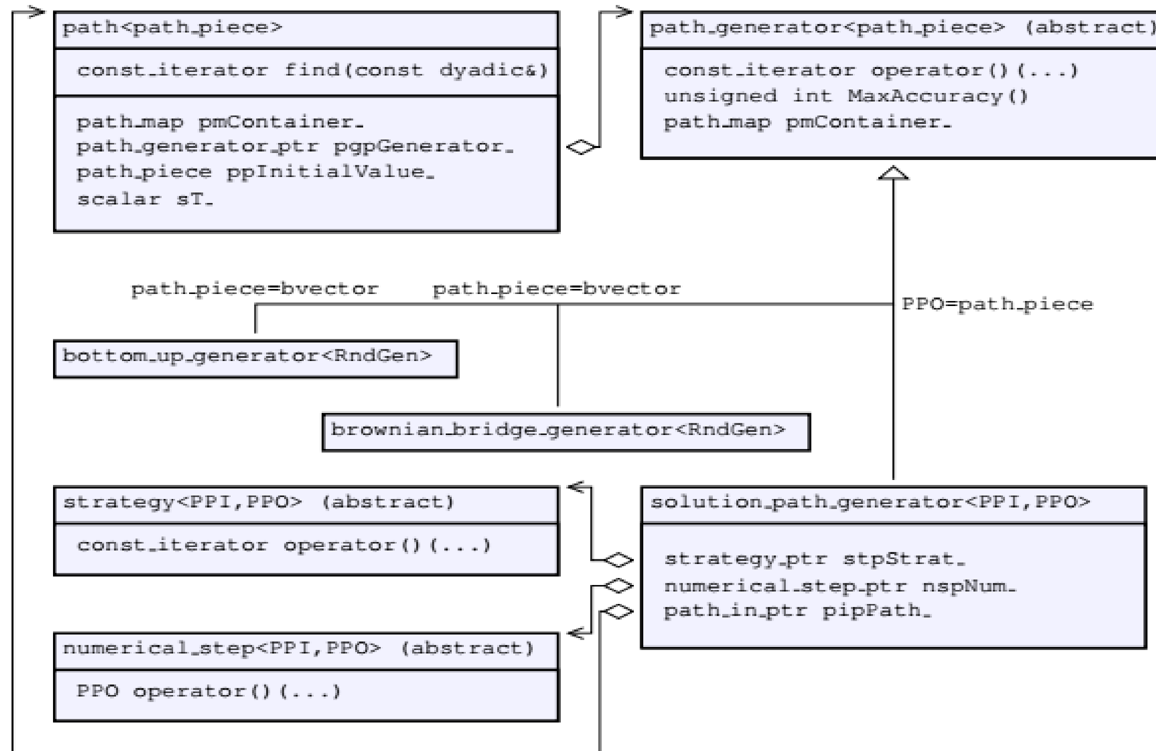
Types defined or used by the MClib

The following **template parameters** are used in the library

- `path_piece` - assumed to be compatible with stl containers. In the particular examples `bvector` is used (see later).
- `PPI`, `PPO` usually refer to input noise `path_piece`'s and `path_piece`'s of a solution to a certain SDE respectively
- `PO` refers to "path out", i.e. solution path of an SDE

The composition of classes in MClib

p.t.o.



Note the separation of responsibilities.

Using MClib

Typical users only work with `mc_factory` and `path<path_piece>` and potentially

- implement particular `payoff` classes or their own approach to computing payoffs,
- implement particular `statistics` classes or implement their own approach to statistics,
- implement Monte Carlo algorithms at high level.

More advanced users implement

- particular `path_generator`'s (`bottom_up_generator` is provided by MClib),
- particular `strategy`'s (`equal_step_strategy` is provided by MClib)
- particular `numerical_step`'s (`euler_numerical_step<VF>` is provided by MClib, where `VF` is the dummy type of a container of coefficient functions defining the SDE).
- particular `mc_factory`'s assembling new MC tools from the new components (one particular is provided by MClib).

The structure shown on the previous slide is **not modifiable** but **extendible**.

Next lecture

High-level interface examples

- estimating strong approximation error
- implementing multilevel Monte-Carlo

MClib lower level examples

- Path dependent payoffs
- General framework for knock-out payoffs
- Particular knock-out conditions
- Quantile statistics
- Brownian bridge generator