ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computational Engineering

Spring semester 2014

Prof. P. Koumoutsakos, G. Tauriello
ETH Zentrum, CLT F 12, C 11
CH-8092 Zürich

Solution 9

Issued: 27.5.2014

Question 1: Rectangle rule

The code is short and straightforward. At a fixed $N = 32$ we get a relative error of about 3%, which is very reasonable. Varying N we observe linear decrease of the error, this agrees with the theoretical result.

```cpp
/*
 *  Created by Gerardo Tauriello.
 *  Copyright 2013 ETH Zurich. All rights reserved.
 */

#include <iostream>
#include <cassert>
#include <vector>
#include <cmath>

using namespace std;

// N = # grid cells -> N+1 grid points, h = 1/N

int main()
{
    // init data
    const int N = 32;
    vector<double> v(N+1);
    const double h = 1./N;
    for (int i = 0; i <= N; ++i) {
        const double x = i*h;
        v[i] = sin(x);
    }
    // integrate
    double res = 0;
    for (int i = 0; i < N; ++i) {
        res += v[i]*h;
    }
    // report
    cout << "Integration result: " << res << endl;
    cout << "Error: " << abs(res - (1-cos(1))) << endl << endl;

    return 0;
}
```

Listing 1: IntegrateQ1.cpp

Question 2: Refactor code

The code listed in this question is an intermediate step between the previous version and the version implementing strategy pattern. This step can be easily omitted, but it can be helpful in figuring out which parts of the code should be wrapped in abstract classes, and what should be implemented in inherited classes. Also it is easier to carry out the convergence study with this version.

```
1  /*
2   *  Created by Gerardo Tauriello.
3   *  Copyright 2013 ETH Zurich. All rights reserved.
4   */
5
6  #include <iostream>
7  #include <cassert>
8  #include <vector>
9  #include <cmath>
10
11 using namespace std;
12
13 // N = # grid cells -> N+1 grid points, h = 1/N
14
15 // Classes implementing Rectangle rule and Sine initialization
16 // No inheritance
17 class SinInitializerSimple {
18 public:
19     void init(int N, vector<double>& v) {
20         v.resize(N+1);
21         const double h = 1./N;
22         for (int i = 0; i <= N; ++i) {
23             const double x = i*h;
24             v[i] = sin(x);
25         }
26     }
27 };
28
29 class RectangleIntegratorSimple {
30 public:
31     double integrate(const vector<double>& v) {
32         double res = 0;
33         const int N = v.size()-1;
34         const double h = 1./N;
35         for (int i = 0; i < N; ++i) {
36             res += v[i]*h;
37         }
38         return res;
39     }
40 };
41
42 int main()
43 {
44     RectangleIntegratorSimple rIntegrator;
45     SinInitializerSimple      sInitializer;
46     vector<double> u;
47     const int Ns[] = {16,32,64,128};
48     const int NNs = 4;
49     for (int i = 0; i < NNs; ++i) {
```

```
50          sInitializer.init(Ns[i], u);
51          double res = rIntegrator.integrate(u);
52          cout << "Error at N = " << Ns[i] << ": " << abs(res - (1-cos(1)))
                << endl;
53      }
54
55      return 0;
56  }
```

Listing 2: IntegrateQ2.cpp

Question 3: Adding more methods

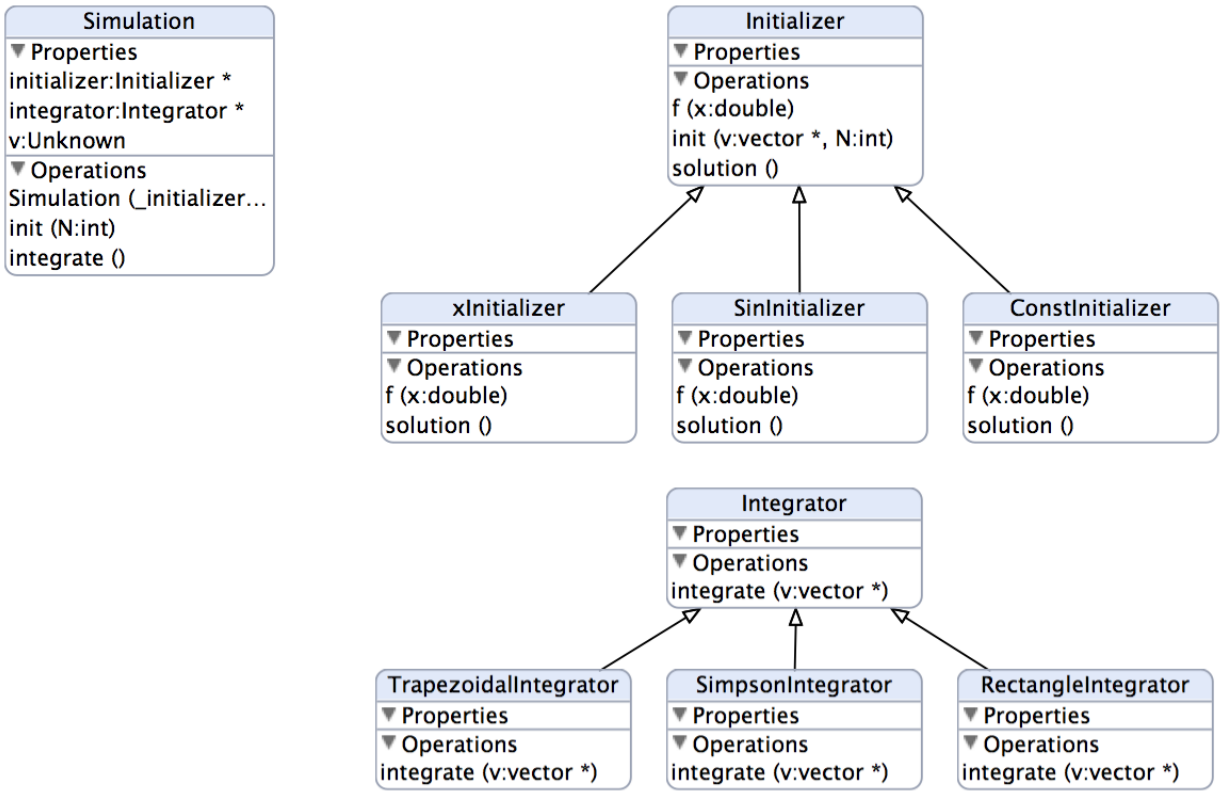

Figure 1: Class diagram of `IntegratorQ3.cpp`

You can see the class diagram of the code implementing strategy pattern in fig. 1.

Initializing loop is implemented in the base class `Initializer`, while the function `f(double x)` itself and analytical solution `solution(double x)` are virtual and defined only in inherited classes. Note that if working with function which doesn't have any analytical representation and only defined on grid points, it would be necessary to redesign `Initializer`.

`Integrator` class doesn't contain any code while only defining an interface. The reason for such design is that the loops of all three different rules are quite different and the common code is small and cumbersome to factor out.

Convergence results shown in fig. 2 (for sine function) confirm theoretical predictions. Note that for const function all methods are exact. For linear function Trapezoidal rule and Simpson's rule

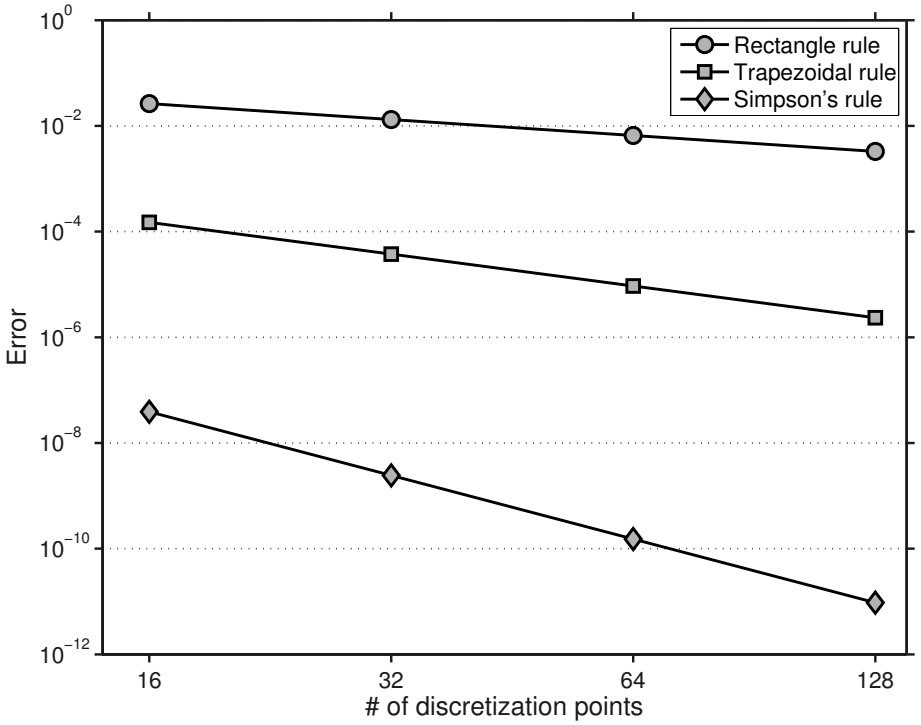

Figure 2: Error convergence with respect to number of discretization points N. Sine function was used here. Note the log-log scale of the plot axis that makes it easy to see the order of accuracy of the methods

are exact, as they use first and second order polynomial approximation correspondingly, while the Rectangle rule is not exact.

```
/*
 *  Created by Gerardo Tauriello.
 *  Copyright 2013 ETH Zurich. All rights reserved.
 */

#include <iostream>
#include <cassert>
#include <vector>
#include <cmath>
#include <string>

using namespace std;

// N = # grid cells -> N+1 grid points, h = 1/N

// Abstract Initializer and specifications
// Abstract Integrator and specifications
// Simulation as Strategy pattern
class Initializer {
public:
    virtual double f(double x) = 0;
    virtual double solution() = 0;

    void init(vector<double>& v, int N) {
```

```
25          double h = 1.0 / N;
26          v.resize(N+1);
27          for (int i = 0; i <= N; ++i) {
28              const double x = i*h;
29              v[i] = f(x);
30          }
31      }
32  };
33  class ConstInitializer: public Initializer {
34  public:
35      virtual double f(double x) { return 1; }
36      virtual double solution()  { return 1; }
37  };
38  class xInitializer: public Initializer {
39  public:
40      virtual double f(double x) { return x; }
41      virtual double solution()  { return 0.5; }
42  };
43  class SinInitializer: public Initializer {
44  public:
45      virtual double f(double x) { return sin(x); }
46      virtual double solution()  { return 1 - cos(1); }
47  };
48
49  class Integrator {
50  public:
51      virtual double integrate(vector<double>& v) = 0;
52  };
53  class RectangleIntegrator: public Integrator {
54      virtual double integrate(vector<double>& v) {
55          double res = 0;
56          const int N = v.size()-1;
57          const double h = 1./N;
58          for (int i = 0; i < N; ++i) {
59              res += v[i]*h;
60          }
61          return res;
62      }
63  };
64  class TrapezoidalIntegrator: public Integrator {
65      virtual double integrate(vector<double>& v) {
66          double res = 0;
67          const int N = v.size()-1;
68          const double h = 1./N;
69          for (int i = 0; i < N; ++i) {
70              res += (v[i] + v[i+1])*h*0.5;
71          }
72          return res;
73      }
74  };
75  class SimpsonIntegrator: public Integrator {
76      virtual double integrate(vector<double>& v) {
77          double res = 0;
78          const int N = v.size()-1;
79          assert((N % 2) == 0);
80          const double h = 1./N;
```

```
81          for (int i = 0; i < N; i+=2) {
82              res += (v[i] + 4*v[i+1] + v[i+2])*h*1.0/3;
83          }
84          return res;
85      }
86  };
87
88  class Simulation {
89      Initializer* initializer;
90      Integrator* integrator;
91      vector<double> v;
92  public:
93      Simulation(Initializer* _initializer, Integrator* _integrator)
94      : initializer(_initializer), integrator(_integrator)
95      {
96      }
97      void init(int N) { initializer->init(v,N); }
98      double integrate() { return integrator->integrate(v); }
99  };
100
101 int main()
102 {
103     const int Ns[] = {16,32,64,128};
104     const int NNs = 4;
105
106     int initializeVariant = -1, integrateVariant = -1;
107
108     while (true)
109     {
110         cout << "Enter 1 for const function, 2 for linear function, 3 for
                sine function: ";
111         cin >> initializeVariant;
112
113         if (1 <= initializeVariant && initializeVariant <= 3) break;
114         cout << "Expected integer value between 1 and 3!" << endl;
115         cin.clear();
116         cin.ignore();
117     }
118     while (true)
119     {
120         cout << "Enter 1 for rectangle rule, 2 for trapezoidal rule, 3
               for Simpson's rule: ";
121         cin >> integrateVariant;
122
123         if (1 <= integrateVariant && integrateVariant <= 3) break;
124         cout << "Expected integer value between 1 and 3!" << endl;
125         cin.clear();
126         cin.ignore();
127     }
128
129     Integrator*  integrator;
130     Initializer* initializer;
131     string initStr, integrStr;
132
133     switch (initializeVariant)
134     {
```

```
135         case 1: initializer = new ConstInitializer();
136                 initStr     = "const function";
137             break;
138 
139         case 2: initializer = new xInitializer();
140                 initStr     = "linear function";
141             break;
142 
143         case 3: initializer = new SinInitializer();
144                 initStr     = "sine function";
145             break;
146     }
147 
148     switch (integrateVariant)
149     {
150         case 1: integrator = new RectangleIntegrator();
151                 integrStr  = "Rectangle rule";
152             break;
153 
154         case 2: integrator = new TrapezoidalIntegrator();
155                 integrStr  = "Trapezoidal rule";
156             break;
157 
158         case 3: integrator = new SimpsonIntegrator();
159                 integrStr  = "Simpson's rule";
160             break;
161     }
162 
163     cout << endl << integrStr << ", " << initStr << endl;
164     Simulation fintegrator(initializer, integrator);
165     for (int i = 0; i < NNs; ++i) {
166         fintegrator.init(Ns[i]);
167         double res = fintegrator.integrate();
168         cout << "Error at N = " << Ns[i] << ": " << abs(res - initializer
              ->solution()) << endl;
169     }
170 
171     delete initializer;
172     delete integrator;
173     return 0;
174 }
```

Listing 3: IntegrateQ3.cpp