Abstractions and Object-Orientation for PDE Solvers

Knut–Andreas Lie

Dept. of Informatics, University of Oslo

Abstractions for PDE Solvers

For PDEs, the discretization consists of

- The unknown u as a scalar or vector *field*
- The field is defined over a *grid*

Primarily a field consists of values at discrete grid points, but may also contain interpolation rules (e.g., for FEM methods).

- Industry PDE simulators: 50 000+ code lines, but not all is number crunching
- Maintainability important
- Should be easy to extend
- Should be easy to use/understand
- Abstractions close to mathematical language are needed

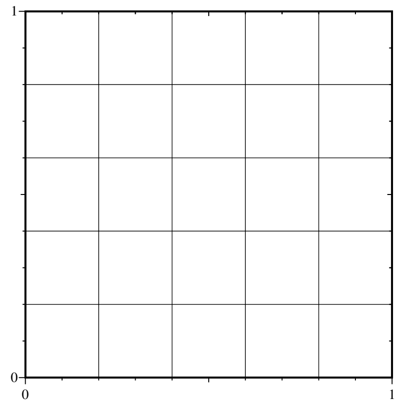
Grid and Field Abstractions

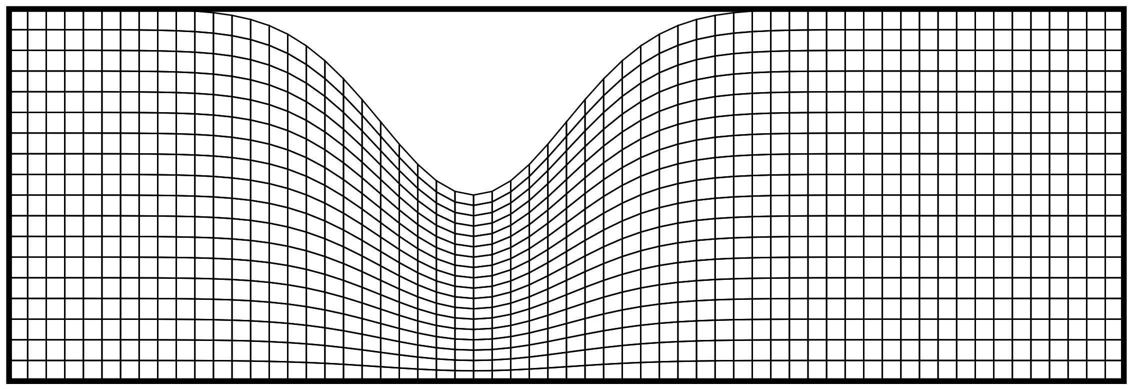
- Consider the PDE (describing e.g., heat conduction):

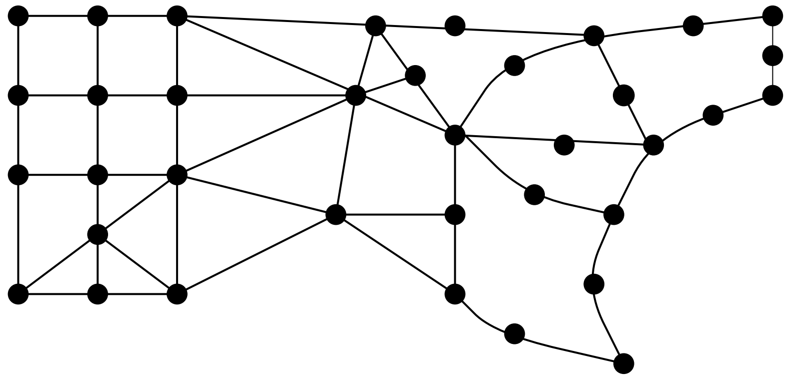
$$-\nabla \cdot [\lambda(\mathbf{x})\nabla u(\mathbf{x})] = f(\mathbf{x}), \quad \mathbf{x} \in \Omega$$

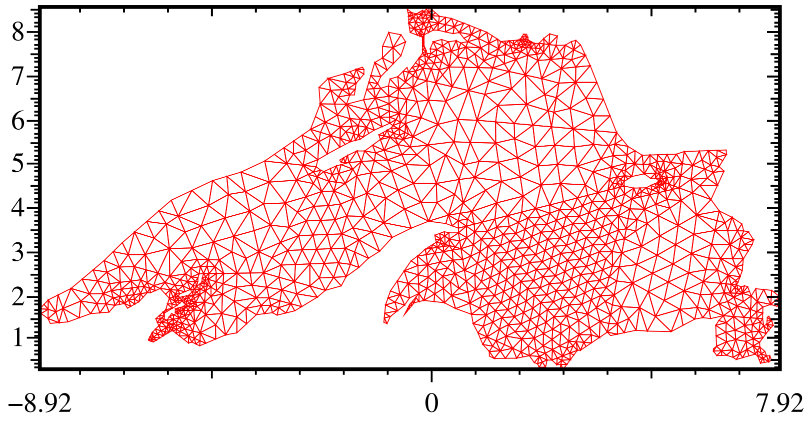
- Principal mathematical quantities:
  - scalar fields: $\lambda(\mathbf{x})$, $u(\mathbf{x})$, $f(\mathbf{x})$
  - domain: Ω
- Principal numerical quantities:
  - scalar fields, arbitrary representation: $\lambda(\mathbf{x})$, $f(\mathbf{x})$ (explicit functions, discrete fields)
  - scalar fields, finite difference type: discrete u
  - grid: discrete Ω

How Does a Grid Look Like?









Programming Considerations

- Obvious ideas:
  - collect grid information in a grid class
  - collect field information in a field class
- Gain:
  - shorter code, closer to the mathematics
  - finite difference methods: minor
  - finite element methods: important
  - big programs: fundamental
  - possible to write code that is (almost) independent of the number of space dimensions (i.e., easy to go from 1D to 2D to 3D!)

Grids and Fields for FDM

Assume a finite difference method (FDM):

- Field represented by `FieldLattice`:
  - a grid of type `GridLattice`
  - a set of point values, `MyArray`
  - `MyArray` is a class implementing user-friendly arrays in one and more dimensions (i.e., an extension of our earlier `MyArray`)
- Grid represented by `GridLattice`
  - lattice with uniform partition in d dimensions
  - initialization from input string, e.g.,

```
d=1 domain: [0,1], index [1:20]

d=3 [0,1]x[−2,2]x[0,10]
    indices [1:20]x[−20:20]x[0:40]
```

Programming with a `GridLattice` Class

```
GridLattice g;    // declare an empty grid
g.scan("d=2␣[0,1]x[0,2]␣[1:10]x[1:40]");   //  initialize  g
const int i0 = g.getBase(1);   //  start  of  first  index
const int j0 = g.getBase(2);   //  start  of second index
const int in = g.getMaxI(1);   // end of  first  index
const int jn = g.getMaxI(2);   // end of second index
int  i,j;
for (i = i0; i <= in; ++i) {
    for (j = i0; j <= jn; ++j) {
        std::cout << "grid␣point␣(" << i << ',' << j
                  << ")␣has␣coordinates␣(" << g.getPoint(1,i)
                  << ',' << g.getPoint(2,j) << ")\n";
    }

// other tasks:
const int nx = g.getDivisions(1);   // number of grid cells  in  x  dir
const int ny = g.getDivisions(2);   // number of grid cells  in  y  dir
const int dx = g.Delta(1);          // grid spacing in  x  dir
const int dy = g.Delta(2);          // grid spacing in  y  dir
```

The GridLattice Class

Data representation:

- Trivial in our case, use: $x_{\min}, x_{\max}, y_{\min}, y_{\max}$, and n_x, n_y.
- For unstructured grids: a carefully designed data structure is vital to obtain code efficiency

```
class GridLattice
{
    static  const int MAX_DIMENSIONS = 2;

    double min[MAX_DIMENSIONS];   //  min coordinate values in  each dimension
    double max[MAX_DIMENSIONS]; // max coordinate values in each dimension
    int   division [MAX_DIMENSIONS]; // number of points in each dimension
    int  dimensions;                 //  number of dimensions
```

The `static` keyword means that `MAX_DIMENSIONS` is a "global" constant shared by all `GridLattice` objects.

The `GridLattice` Class, cont'd

Member functions:

- Constructors
- Initialization – through the `scan` function
- Accessors – access to internal data structure

```
public:
    GridLattice ();
    GridLattice(int  nx, int  ny, double xmin_, double xmax_,
                     double ymin_, double ymax_);

    int  getNoSpaceDim () const;

    double xMin(int dimension) const;
    double xMax(int dimension) const;

    int  getDivisions(int  i ) const; //  get  the  number of points in  each dimension
    int  getNoPoints() const;         //  get the  total  number of points in  the  grid
```

The `GridLattice` Class, cont'd

```
        double Delta(int dimension) const;
        double getPoint(int dimension, int index);

        int getBase(int dimension) const;    // get base values
        int getMaxI(int dimension) const;    // upper limit of array

        void scan(const std::string& init_string );    // scan parameters from string

        friend std :: ostream& operator<<(std::ostream&, GridLattice&);
    };
```

Mutators, i.e., functions for setting internal data members, are not implemented here. Examples: `setBase(..)`, `setMaxI(..)`, etc.

The `friend` keyword enables `operator`« to access private data in the `GridLattice` object.

The `GridLattice` Class, cont'd

```
double GridLattice:: xMin(int dimension) const
{ return min[dimension - 1]; }

double GridLattice:: xMax(int dimension) const
{ return max[dimension - 1]; }

inline int GridLattice :: getDivisions(int i) const
{ return division[i-1]; }

int GridLattice :: getNoPoints() const {
    int return_value = 1;
    for(int i = 0; i != dimensions; ++i)
        return_value *= division[i];
    return return_value;
}
```

Inline functions means that the function call can be optimized away by the compiler.

The GridLattice Class, cont'd

Nested inline functions:

```
inline double GridLattice:: Delta(int dimension) const {
  return (max[dimension−1] − min[dimension−1]) / double(division[dimension−1]);
}
inline double GridLattice:: getPoint(int dimension, int index) {
    return min[dimension−1] + (Delta(dimension) ∗ (index − 1));
}
```

Some compilers do not allow nested inlines. To come around this, we can use a preprocessor macro:

```
#ifdef NO_NESTED_INLINES
    return min[dimension−1] + ((max[dimension−1]− min[dimension−1])
                 / double(division[dimension−1]))∗(index − 1);
#else
    return min[dimension−1] + (Delta(dimension) ∗ (index − 1));
#endif
```

The `GridLattice` Class, cont'd

Typical call: `g.scan("d=2 [0,1]x[0,2] [1:10]x[1:40]");`

```
void GridLattice :: scan(const string& init_string )
{
    using namespace std; // allows dropping std :: prefix
    istrstream is( init_string .c_str ());

    // ignore "d="
    is.ignore(1, 'd');   is.ignore(1, '=');

    // get the dimensions
    is >> dimensions;
    if (dimensions < 1 || dimensions > MAX_DIMENSIONS) {
        cerr << "GridLattice::scan() -- illegal dimensions "
             << dimensions << endl;
        cerr << "   MAX_DIMENSIONS is set to "
             << MAX_DIMENSIONS << endl;
        exit (1);
    :
```

The `GridLattice` Class, cont'd

```
GridLattice :: GridLattice(int nx, int ny, double xmin, double xmax,
                                    double ymin, double ymax) {
    dimensions = 2;
    max[0] = xmax;      max[1] = ymax;
    min[0] = xmin;      min[1] = ymin;
    division [0] = nx;  division [1] = ny;
}

GridLattice :: GridLattice () {
    dimensions = 2;
    int i;
    for (i = 1; i <= MAX_DIMENSIONS; ++i) {
        min[i ] = 0;  max[i ] = 1;  division [ i ] = 2;
    }
}
```

We have chosen to initialize internal data also for the empty constructor to avoid errors.

The `FieldLattice` Class

```
class FieldLattice {
protected:
    Handle<GridLattice>        grid_lattice ;
    Handle< MyArray<real> > grid_point_values;
    std :: string                    fieldname;
public:
    // make a field from a grid and a fieldname:
    FieldLattice (GridLattice& g, const std:: string& fieldname);

          MyArray<real>& values()          { return *grid_point_values; }
    const MyArray<real>& values() const { return *grid_point_values; }

          GridLattice& grid ()                   { return * grid_lattice ; }
    const GridLattice& grid () const          { return * grid_lattice ; }

    std :: string  name() const                  { return fieldname; }

    void values(MyArray<real>& new_array);
};
```

The `FieldLattice` Class, cont'd

- Inline functions are obtained by implementing the function body inside the class declaration.
- We use a parameter `real`, which equals `float` or `double` (by default).
- The `Handle<>` construction is a *smart pointer*, implementing reference counting and automatic deallocation (garbage collection).
- Using a `Handle<GridLattice>` object instead of a `GridLattice` object, means that a grid can be shared among several fields.

  Example: For the wave equation `um`, `u`, and `up` may share the same grid (and possibly also `Hm`, `H`, and `Hp`).

Smart Pointers

Observations:

- Dynamic memory in C/C++ means that pointers are needed
- Lack of garbage collection (automatic clean-up of memory that is no longer in use) means that manual deallocation is important
- Codes with memory leakage slowly eat up the memory and slow down computations
- How does one determine when memory is no longer in use?

  Example: Suppose 5 fields point to the same grid — when can we safely remove the grid object?
- *Pointers are bug no. 1 in C/C++ ...*

Smart Pointers — Reference Counting

Solution to problems:

- Avoid explicit deallocation
- Introduce reference counting: count the number of times a smart pointer “uses a pointer” (i.e., points to a given object).

Advantages:

- negligible overhead
- automatic garbage collection
- several fields can safely share one grid

The `FieldLattice` Class, cont'd

```
FieldLattice :: FieldLattice (GridLattice& g, const std::string& name_)
{
  grid_lattice .rebind(&g);

  // allocate the grid_point_values array:
  if ( grid_lattice ->getNoSpaceDim() == 1)
    grid_point_values.rebind(new MyArray<real>(grid_lattice->getDivisions(1)));
  else if ( grid_lattice ->getNoSpaceDim() == 2)
    grid_point_values.rebind(new MyArray<real>(
        grid_lattice ->getDivisions(1), grid_lattice ->getDivisions(2)));
  else
    ; // three-dimensional fields are not yet supported...
  fieldname = name_;
}

void FieldLattice :: values(MyArray<real>& new_array)
{ grid_point_values.rebind(&new_array); }
```

Simulator Classes

- The PDE solver is a class itself
  - ⇒ easy to combine solvers (systems of PDEs)
  - ⇒ easy to extend/modify solver
  - ⇒ enables coupling to optimization, automatic parameter analysis etc.
- Typical look (for a static problem):

```
class MySim
{
protected:
    // grid and field objects
    // PDE dependent parameters
public:
  void scan();             // read input and init
  void solveProblem();
  void resultReport();
};
```

Heat.C Revisited

```
#ifndef Heat1D1_h_IS_INCLUDED
#define Heat1D1_h_IS_INCLUDED
:
class Heat1D1
{
protected:                          // data items visible in subclasses
   MatTri<real>           A;        // the coefficient matrix
   MyArray<real>          b;        // the right-hand side
   Handle<GridLattice> grid;        // 1D grid
   FieldLattice           u;        // the discrete solution
   real                   alpha;    // parameter in the test problem
public:
   Heat1D1() {}
  ~Heat1D1() {}
   void scan ();                  // read input, set size of A, b and u
   void solveProblem ();          // compute A, b; solve Au=b
   void resultReport ();          // write and plot results
};
#endif
```

Reading Input

```
void Heat1D1:: scan ()
{
  char gridstr [30];
  int n;   // no of intervals in the domain (0,1)

  CommandLineArgs::read("-n", n, 5); // e.g ., ./ heat1d -n 20

  grid.rebind (new GridLattice());
  sprintf ( gridstr ,"d=1␣[0,1]␣[1:%d]",n);
  grid->scan(gridstr);

  u.rebind (new GridLattice(*grid, "u" ));
  A.redim(n);              // set size of tridiagonal matrix A
  b.redim(n);              // set size of vector b
  CommandLineArgs::read ("-a", alpha, 0.0);
}
```

`CommandLineArgs::read` is described closer in [R&L]

Solving the Problem

```
void Heat1D1:: solveProblem () {
  A. fill  (0.0);   b. fill  (0.0);
  const int   n = b.size ();    //  alternative :  grid−>getMaxI(1)
  const real h = grid−>Delta(1);
  real  x ;  int  i ;

  i  = 1;  A (1,1) = 1;  A (1,2) = 0;  b (1) = 0;
  for  ( i  = 2;  i  <  n ;  i ++) {
    x  =  grid−>getPoint(1,i);
    A(i−1,i ) = 1;   A(i , i ) = −2;   A(i , i +1) = 1;
    b(i ) =  h∗h∗(alpha+1)∗pow(x,alpha);
  }
  x  =  grid−>getPoint(1,n);
  A(n−1,n) = 2;  A(n,n) = −2;
  b(n) = − 2∗h + h∗h∗(alpha+1)∗pow(x,alpha);

  A.factLU ();
  A.forwBack(b,u.values());
}
```

The Main Program

```
#include <Heat1D1.h>

int main(int argc, const char* argv[])
{
  Heat1D1 simulator;
  simulator.scan ();
  simulator.solveProblem ();
  simulator.resultReport ();
  return 0; // success
}
```

So far, very little has been gained, but let us also consider the nonlinear case

Nonlinear Heat Conduction

- Heat conduction typically depends upon the temperature

$$-\frac{d}{dx}\left(\lambda(u)\frac{du}{dx}\right) = f(x), \quad 0 < x < 1$$

  This is a *nonlinear differential equation*.

- Straightforward discretization gives

$$\lambda(u_{i+\frac{1}{2}})(u_{i+1} - u_i) - \lambda(u_{i-\frac{1}{2}})(u_i - u_{i-1}) = -h^2 f_i$$

- Since $\lambda(u_{i+\frac{1}{2}})$ will depend upon u_i and u_{i+1}, this is a set of *nonlinear algebraic equations*

$$\mathbf{A}(\mathbf{u})\mathbf{u} = \mathbf{b}(\mathbf{u})$$

  i.e., the coefficients are unknown quantities....

Sucessive Substitution

Idea: use the following simple algorithm

```
Guess a solution $\mathbf{u}^0$.
Repeat
    solve $\mathbf{A}(\mathbf{u}^n)\mathbf{u}^{n+1} = \mathbf{b}(\mathbf{u}^n)$
until difference of $\mathbf{u}^n$ and $\mathbf{u}^{n+1}$ is small
```

Each equation is now a *linear* equation with variable coefficients.

Advantage: may reuse previous code by inserting evaluations of the coefficient $\lambda(u_{i+\frac{1}{2}})$

Disadvantage: slow convergence

Implementation

Reuse old program `HeatTri` with:

- Loop around system generation and solution
- Two arrays `uk` and `ukm`
- Initial guess in `ukm`
- New auxiliary variables (for iteration etc.)
- Function `lambda` to evaluate $\lambda(u)$
- Update `A[mcp]` and `b` for each step
- Check for termination upon convergence
- Set `ukm` equal `uk` before new iteration

Implementation in Pseudocode

```
// Iteration loop
Repeat
    // Assemble matrix and left−hand side
    for i=1:n
        // Left boundary
        // Right boundary

        // Interior
        else
            l1 = lambda( u(i−1) );
            l2 = lambda( u(i)   );
            l3 = lambda( u(i+1) );
            A(i,i−1) =  0.5*(l1 + l2 );
            A(i,i)   = −0.5*(l1 + 2*l2 + l3 );
            A(i,i−1) =  0.5*(l2 + l3 );
    end

    // Solve linear system
```

Nonlinear Heat Conduction, cont'd

We can implement different $\lambda()$ functions objects (or *functors*) in a class hierarchy.

First define the base class:

```
class LambdaFunc : public HandleId
{
public:
   virtual  real  lambda (real u)=0;
   virtual  real  exactSolution ( real  x)=0;
   virtual  void scan () =0;
   virtual  string  formula ()=0;
};
```

No data and all functions pure virtual, thus defining only *an interface* (providing access all functions).

Nonlinear Heat Conduction, cont'd

Then a particular realisation in a subclass:

```
class Lambda1 : public LambdaFunc
{
   real  m;
public:
   Lambda1() {}
   virtual  real  lambda (real u)
             {  return pow(u,m); }
   virtual  real  exactSolution ( real  x)
             {  return pow(x,1/(m+1)); }
   virtual  void scan ()
             {  CommandLineArgs::read("-m",m,0.0); }
   virtual  String  formula  () {  return "u^m"; }
};
```

Notice that the parameter m is a member of the *subclass* and not of the base class

The `Heat1D_NL` Class

```
class Heat1D_NL {
protected:                                  // data items visible in subclasses
  MatTri<real>           A;                 // the coefficient matrix
  MyArray<real>          b;                 // the right−hand side
  Handle<GridLattice> grid;                 // 1D grid
  FieldLattice           uk;                // the discrete solution u^k
  FieldLattice           ukm;               // the discrete solution u^{k−1}
  real                   epsilon;           // tolerance in nonlinear iteration
  Handle<LambdaFunc> lambda; // specific lambda function
public:
  Heat1Dn1() {}
 ~Heat1Dn1() {}
  void scan  ();                         // read input, set size of A, b, and u
  void makeSystem();                     // compute A, b;
  void solveSystem();                    // solve Au^k=b
  void solveProblem ();                  // nonlinear iteration loop
  void resultReport  ();                 // write numerical error ( if possible)
};
```

Using λ in the Code

Picking a specific realisation

```
int lambda_tp;
CommandLineArgs::read("-N", lambda_tp, 1);
if (lambda_tp == 1)
  lambda.rebind (new Lambda1());
else if (lambda_tp == 2)
  lambda.rebind (new Lambda2());
else
```

Initialize function-specific parameters: `lambda->scan();`

Evaluating λ:

```
lambda->lambda(ukm.values()(i-1));
```

Computing the exact solution (if available):

```
lambda->exactSolution(grid->getPoint(1,i));
```

Wave Equation Revisited

We wish to solve:

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot \big(H(x)\nabla u\big), \qquad u(x,0) = I(x), \quad u_t(x,0) = 0$$

The solution algorithm was presented in Lecture 4.
What are the natural objects here?

- a grid object of type `GridLattice`
- three fields of type `FieldLattice` for $u^+_{i,j}$, $u_{i,j}$, and $u^-_{i,j}$
- a field of type `FieldLattice` for the depth $H_{i,j}$
- a time integration parameter object `TimePrm`
- functor hierarchies[1] for the known functions $H(x)$ and $I(x)$

Working with class hierarchies and virtual functions is really the point that qualifies this to be called object-oriented programming as opposed to “programming with objects”...

The TimePrm Class

```
class TimePrm
{
    double time_;     // current time value
    double delta;     // time step size
    double stop;      // stop time
    int    timestep;  // time step counter
public:
    TimePrm(double start, double delta, double stop)
        : time_(start ), delta(delta ), stop(stop)
    { initTimeLoop(); }

    double time()          { return time_; }
    double Delta()         { return delta ; }
    void initTimeLoop()    { time_ = 0; timestep = 0; }
    bool finished ()       { return (time >= stop) ? true : false ; }
    void increaseTime() { time_ += delta; ++timestep ; }
    int getTimeStepNo() { return timestep; }
};
```

The Wave2D1 Class

```
class Wave2D1 {
    Handle<GridLattice> grid ;          // lattice grid here 1D grid
    Handle<FieldLattice> up,u,um; // solution u at time levels l+1, l , and l−1
    Handle<FieldLattice> lambda;  // variable coefficient (depth)
    Handle<FieldLattice> tmp;     // variable coefficient (depth)
    Handle<TimePrm>   tip ;       // time integr . prms: dt, t_stop
    Handle<WaveFunc> H, I ;       // function for depth and initial surface

    void timeLoop();              // perform time stepping
    void plot(bool initial );     // dump fields to file , plot later
    void WAVE(FieldLattice& up, const FieldLattice& u,
              const FieldLattice& um, real a, real b, real c);
    void setIC ();                // set initial conditions
    void setH ();                 // load H into lambda for efficiency
    real calculateDt(int func);   // calculate optimal timestep
public:
    void scan();                  // read input and initialize
    void solveProblem();          // start the simulation
};
```

The Wave2D1 Class, cont’d

```
void Wave2D1:: solveProblem () {
  setIC ();              // set initial conditions
  timeLoop();            // run the algorithm
}

void Wave2D1:: setIC () {
  const int nx = grid−>getMaxI(1);
  const int ny = grid−>getMaxI(2);

  // set initial surface elevation
  MyArray<real>& uv = u−>values();
  for ( int j = 1; j <= ny; j++)
      for ( int i = 1; i <= nx; i++)
          uv(i , j ) = I−>valuePt(grid−>getPoint(1, i), grid−>getPoint(2, j ));

  // set the help variable um:
  WAVE (∗um, ∗u, ∗um, 0.5, 0.0, 0.5);
}
```

The Wave2D1 Class, cont'd

```
void Wave2D1:: timeLoop ()
{
  tip −>initTimeLoop();
  plot (true);

  while(! tip −>finished ()) {
       tip −>increaseTime();

       WAVE (∗up, ∗u, ∗um, 1, 1, 1);

       // move handles (get ready for next step):
       tmp = um; um = u; u = up; up = tmp;

       plot (false);
  }
}
```

The Wave2D1 Class, cont'd

```
void Wave2D1:: scan () {
  // create the grid ...
  grid.rebind(new GridLattice());
  grid->scan(CommandLineArgs::read("-grid",
       "d=2␣[-10,10]x[-10,10]␣[1:30]x[1:30]"));
  cout << (*grid) << endl;

  // create new fields ...
  up.     rebind(new FieldLattice(*grid, "up"));
  u.      rebind(new FieldLattice(*grid, "u"));
  um.     rebind(new FieldLattice(*grid, "um"));
  lambda.rebind(new FieldLattice(*grid, "lambda"));

   // select the appropriate I and H
   int func = CommandLineArgs::read("-func", 1);
   if (func == 1) {
       H.rebind(new GaussianBell('H'));
       I.rebind(new GaussianBell('U'));
   }
   else {
       H.rebind(new Flat());
       I.rebind(new Plug('U'));
   }

   // initialize the parameters in the functions.
   H->scan();
   I->scan();

   // set H field and compute optimal dt
   setH();
   tip.rebind(new TimePrm(0, calculateDt(func),
       CommandLineArgs::read("-tstop", 30.0)));
}
```

Object-Oriented Implementation of FEMs

Very much to be gained from using object orientation (e.g., as in Diffpack):

- based on generic FEM tools for
  - coordinate mappings
  - numerical integration
  - assembling of linear systems
  - solution of linear systems
- implement only problem specific part
- heavy code reuse
- inheritance and virtual functions are crucial
- great flexibility, e.g., code can be made independent of number of space dimensions

Example: the Poisson Equation

Model problem: $-\nabla \cdot [k(x)\nabla u(x)] = f(x), \quad x \in \Omega$

Weak formulation:

$$A_{i,j} = \int_\Omega k(x)\nabla N_i(x)\nabla N_j(x)d\Omega, \quad b_i = \int_\Omega f(x)N_i(x)d\Omega.$$

Elementwise computation in local coordinates

$$A_{i,j}^{(e)} = \int_{\Omega_e} k\nabla N_i \cdot \nabla N_j \, d\Omega_e = \int_{\tilde{\Omega}} k\nabla N_i \cdot \nabla N_j \det J \, d\xi_1 \cdots d\xi_d$$

$$b_i^{(e)} = \int_{\Omega_e} f N_i \, d\Omega_e = \int_{\tilde{\Omega}} f N_i \det J \, d\xi_1 \cdots d\xi_d$$

Numerical integration

$$A_{i,j}^{(e)} \approx \sum_{k=1}^{n_I} I_{i,j}(\xi_k)w_k, \quad b_i^{(e)} \approx \sum_{k=1}^{n_I} K_i(\xi_k)w_k.$$

Diffpack FEM Applications

- FEM solvers in Diffpack are represented as separate classes.
- Such classes are derived from the skeleton application (base class) `FEM`.
- The derived simulator class must provide problem-specific data:
  - the PDE in terms of the discrete integrands $I_{i,j}(\xi)w$ and $K_i(\xi)w$,
  - handling of boundary values,
  - for time-dependent problems: Handling of the initial value.
- These problem-specific details are provided by virtual functions defined by class `FEM`.

The Function `integrands`

```
void integrands (ElmMatVec& elmat, const FiniteElement& fe);
```

- The object `fe` is preloaded (from class `FEM`) with the current integration point ξ_k
- Moreover, it contains N_i, ∇N_i, $\det J \cdot w_k$ and other useful quantities, such as the global coordinates x corresponding to ξ_k
- The user must provide evaluation of the functions $f(x)$ and $k(x)$
- The function `integrands` should add the contributions $I_{i,j} w_k$ and $K_i w_k$ to the *elemental* matrix (`Mat(real)` structure `elmat.A`), and the *elemental* vector (`Vec(real)` structure `elmat.b`)

Example: `Poisson0`

Consider the test problem in 2D, $\Omega = (0,1) \times (0,1)$. The simulator class `Poisson0` consists of:

- `scan` – geometry input, allocates grid and some other large objects.
- `fillEssBC` – sets the essential boundary conditions $u = 0$ on $\partial\Omega_E$.
- `integrands` – samples the element-based integrands. Assembly is handled automatically by `FEM::makeSystem`.
- `solveProblem` – entry point for the simulator.

The Poisson0 Class

```
#ifndef Poisson0_h_IS_INCLUDED
#define Poisson0_h_IS_INCLUDED

#include <FEM.h>            // FEM algorithms, class FieldFE, GridFE
#include <DegFreeFE.h>      // degree of freedom book−keeping
#include <LinEqAdmFE.h>     // linear systems: storage and solution

class Poisson0 : public FEM {
protected:
  // general data:
  Handle(GridFE)     grid;   // pointer to a finite element grid
  Handle(DegFreeFE) dof;    // trivial book−keeping for a scalar PDE
  Handle(FieldFE)    u;      // finite element field, primary unknown
  Vec(real)          linsol; // solution of the linear system
  Handle(LinEqAdmFE) lineq; // linear system: storage and solution

  void fillEssBC ();          // set boundary conditions u=g
  virtual void integrands(ElmMatVec& elmat, const FiniteElement& fe);
```

The `Poisson0` Class, cont'd

```
public:
   Poisson0 ();
  ~Poisson0 () {}

   void scan ();                  // read and initialize data
   void solveProblem ();          // main driver routine
   void resultReport ();          // write comparison with analytical sol.

   real f(real x, real y);        // source term in the PDE
   real k(real x, real y);        // coefficient in the PDE
   real g(real x, real y);        // essential boundary conditions
};

#endif
```

Poisson0::scan

```
void Poisson0:: scan ()
{
   // extract input from the command line:
   int nx, ny;   // number of nodes in x− and y−direction
   initFromCommandLineArg ("−nx", nx, 6); // read nx, default : nx=6
   initFromCommandLineArg ("−ny", ny, 6);
   String elm_tp;
   initFromCommandLineArg ("−elm", elm_tp, "ElmB4n2D");
```

- Read information about the number of nodes in the x- and y-directions.
- Read information about the element type.

Poisson0::scan, cont'd

```
// the box preprocessor requires input on the form (example):
// geometry:  d=2 [0,1]x[0,1]
//  partition :  d=2 elm= ElmB4n2D div=[4,4] grading=[1,1]
String geometry = "d=2␣[0,1]x[0,1]" ;                    // 2D specific
String  partition  = aform("d=2␣elm=%s␣div:[%d,%d]␣grading:[1,1]",
                           elm_tp.c_str (), nx−1,ny−1); // 2D specific

grid.rebind (new GridFE()); // make an empty grid
PreproBox p;                    // preprocessor for box−shaped domains
p.geometryBox() .scan (geometry);  //  initialize  the geometry
p.partitionBox (). scan ( partition );  //  initialize  the  partition
p.generateMesh (∗grid);               // run the preprocessor
```

- Construct input strings to the box preprocessor, specifying the domain, the partition, and the element type.
- Call the box preprocessor to generate the grid.

`Poisson0::scan`, cont'd

```
    u.rebind (new FieldFE (*grid,"u" ));        // allocate , set name="u"
    dof.rebind (new DegFreeFE (*grid, 1)); // 1 unknown per node
    lineq.rebind (new LinEqAdmFE());            // Ax=b system and solvers
    linsol .redim (grid->getNoNodes());         // redimension linsol
    lineq->attach (linsol );                    // use linsol as x in Ax=b

    // banded Gaussian elimination is the default solver in lineq
}
```

- Set `u` to point to a new finite element field over the grid.
- Set `dof` to point to a new `DegFreeFE` object.
- Set `lineq` to point to a new `LinEqAdmFE` object.
- Redimension the unknown vector in the linear system and attach it to `lineq`.

Poisson:FillEssBC

```
void Poisson0:: fillEssBC  ()
{
  dof−>initEssBC ();                         //  init  for assignment below
  const int nno = grid−>getNoNodes(); // no of nodes
  Ptv(real ) x ;                             // a nodal point
  for ( int i = 1; i <= nno; i ++) {
    // is node i subjected to any boundary indicator?
    if ( grid−>boNode (i)) {
      x = grid−>getCoor (i);                 //  extract coor. of node i
      dof−>fillEssBC (i , g(x (1), x (2))); // u=g at boundary nodes
    }
  }
  dof−>printEssBC (s_o, 2);                  // debug output
}
```

Initialize assignment of essential boundary conditions

Loop through all nodes and if a node is on the boundary, set essential boundary condition

`Poisson0::integrands`

The `integrands` function:

1. Evaluate the Jacobian determinant times the integration weight.
2. Find the global coordinates of the current integration point.
3. Evaluate the f and k functions at this global point.
4. For $i = 1$ to the number of basis functions (element nodes)
   - For $j = 1$ to the number of basis functions
     - Add the appropriate value to the *elemental* matrix.
   - Add the appropriate value to the *elemental* vector.

`Poisson0::integrands`, cont'd

```
 void Poisson0::integrands (ElmMatVec& elmat,const FiniteElement& fe)
{
   // find the global coord. xy of the current integration point:
   Ptv(real) xy = fe.getGlobalEvalPt();
   const real x = xy(1);  const real y = xy(2);     // 2D specific
   const real f_value = f(x,y);                      // 2D specific
   const real k_value = k(x,y);                      // 2D specific
   int i,j;
   const int nbf = fe.getNoBasisFunc(); // = no of nodes in element
   const real detJxW = fe.detJxW();       // Jacobian * intgr. weight


   for (i = 1; i <= nbf; i++) {
     for (j = 1; j <= nbf; j++)
       elmat.A(i,j) += k_value*(fe.dN(i,1)*fe.dN(j,1)  // 2D specific
                                + fe.dN(i,2)*fe.dN(j,2))*detJxW;
     elmat.b(i) += fe.N(i)*f_value*detJxW;
   }
}
```

Problem-Dependent Functions

Specific test case:

$$f(x,y) = -2x(x-1) - 2y(y-1),$$
$$g(x,y) = 0, \quad k(x,y) = 1.$$

Analytical solution: $u(x,y) = x(x-1)y(y-1)$

The functions `f`, `g`, and `k` are then:

```
real Poisson0:: f (real x, real y)
{ return -2*x*(x-1)-2*y*(y-1); }

real Poisson0:: g (real /*x*/, real /*y*/)
{ return 0; }

real Poisson0:: k (real /*x*/, real /*y*/)
{ return 1; }
```

Solving the Problem

```
void Poisson0::solveProblem ()  // main routine of class Poisson0
{
   fillEssBC ();                      // set essential boundary conditions
   makeSystem (*dof, *lineq);         // assembly algorithm from class FEM
   linsol.fill (0.0);                 // init start vector (good habit)
   lineq->solve();                    // solve linear system
   dof->vec2field (linsol, *u);       // load solution (linsol) into u
}


int main (int argc, const char* argv[])
{
   initDiffpack (argc, argv);
   Poisson0 simulator;                // make a simulator object
   simulator.scan ();                 // read and initialize data
   simulator.solveProblem ();         // main routine: compute the solution
   simulator.resultReport ();         // compare with exact solution
   return 0;
}
```

The `FEM::makeSystem` algorithm:

```
initialize global linear system:
set $A_{i,j} = 0$ for $i, j = 1, \ldots, n$,        set $b_i = 0$ for $i = 1, \ldots, n$
loop over all elements:
for $e = 1, \ldots, m$
    set $\tilde{A}^{(e)}_{r,s} = 0$, $r, s = 1, \ldots, n_e$,        set $\tilde{b}^{(e)}_r = 0$, $r = 1, \ldots, n_e$
    loop over numerical integration points:
    for $k = 1, \ldots, n_I$
        evaluate $\tilde{N}_r(\xi_k)$, derivatives of $\tilde{N}_r$ wrt. $\xi$ and $x$, $J$
        integrands:
        for $r = 1, \ldots, n_e$
            for $s = 1, \ldots, n_e$
                $\tilde{A}^{(e)}_{r,s} := \tilde{A}^{(e)}_{r,s} + \frac{d\tilde{N}_r}{dx}\frac{\tilde{N}_s}{dx} \det J\, w_k$
            $\tilde{b}^{(e)}_r := \tilde{b}^{(e)} + f(x^{(e)}(\xi_k))N_r \det J\, w_k$
    fillEssBc:
    for $r = 1, \ldots, n_e$
        if node $r$ has an essential boundary condition then
            modify $\tilde{A}^{(e)}_{r,s}$ and $\tilde{b}^{(e)}_r$ due to this condition
    assemble:
    for $r = 1, \ldots, n_e$
        for $s = 1, \ldots, n_e$
            $A_{q(e,r),q(e,s)} := A_{q(e,r),q(e,s)} + \tilde{A}^{(e)}_{r,s}$
        $b_{q(e,r)} := b_{q(e,r)} + \tilde{b}^{(e)}_r$
```

Summary

Concluding remarks on FEM:

- The simulator class provides only functionality that is closely related to the concrete problem.
- This functionality is part of a larger framework implemented in the base class `FEM`.
- The use of inheritance and virtual functions is crucial in this software architecture.

Software abstractions:

- Brings the implementation closer to the mathematics
- Hides unnecessary details
- FDM: nice, but sometimes trivial and obfuscating
- FVM/FEM: important, in particular for unstructured grids

In this course: since we mostly study trivial examples, it might be hard to see the benefit