Example: Partial Differential Equations

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing

Outline

What Do We Have

- Two simple classes for structured grids (`Domain`, `Curvebase`)
- A simple implementation of a matrix class (`Matrix`; don't use it for production codes!)

What Do We Want

- A class for representing grid functions
- Imposing boundary conditions
- A class for solving PDEs

Our running example will be the heat equation in 2D,

$$\frac{\partial}{\partial t}u = \frac{\partial^2}{\partial x^2}u + \frac{\partial^2}{\partial y^2}u.$$

The Domain Class

This is what we have so far:

```
class Domain {
  public:
    Domain(Curvebase&, Curvebase&, Curvebase&,
           Curvebase&);
    void generate_grid(...);
    // more members
  private:
    Curvebase *sides[4];
    // more members
};
```

- We will need additional members for handling grids. *Since grids do not allow any algebraic manipulation, using our `Matrix` class is not appropriate.*
- We will use C-style arrays.
- It might be more convenient to use STL containers (e.g., `vector`).

The Domain Class: Enhanced

```
class Domain {
  public:
    Domain(Curvebase&, Curvebase&, Curvebase&,
           Curvebase&) : m(0), n(0), x(nullptr),
           y(nullptr) {}
    void generate_grid(int m_, int n_);
    int xsize() { return m; }
    int ysize() { return n; }
    Point operator()(int i, int j);
    bool grid_valid() { return m != 0; }
    // more members
  private:
    Curvebase *sides[4];
    int m, n;
    double *x, *y;
    // more members
};
```

One Dimensional Differences 1

- Consider a grid Ω_h,

$$a = x_0 < x_1 < \cdots < x_{n-1} < x_m = b.$$

- Let $h_i = x_i - x_{i-1}$. Then define, for a grid function $u : \Omega_h \to \mathbb{R}$,

$$D_- u_i = \frac{u_i - u_{i-1}}{h_i}$$
$$D_+ u_i = \frac{u_{i+1} - u_i}{h_{i+1}}$$

- If u is the restriction of a smooth function onto Ω_h, these approximations are first order accurate.
- If the grid is equidistant, $D_+ D_-$ is a second order accurate approximation of $u''(x_i)$ and

$$D_+ D_- u_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$$

One Dimensional Differences 2

$$Du_i = \frac{u_{i+1} - u_{i-1}}{2h}$$

- First oder approximation to u' on a general grid
- Second order accuracy on a constant stepsize grid

Boundaries

- The operators introduced above are not applicable at boundaries.
- Possibility 1: One-sided differences

$$Du_0 = \frac{3u_0 - 4u_1 + u_2}{3h}$$
$$Du_m = \frac{u_{m-2} - 4u_{m-1} + 3u_m}{3h}$$

- Possibility 2: Use ghost points

$$Du_0 = \frac{u_1 - u_{-1}}{2h}$$
$$Du_m = \frac{u_{m+1} - u_{m-1}}{2h}$$

How to get values for the ghost points?

Nonuniform Grids

- Order of approximation is determined using Taylor expansions.
- Ansatz:

$$u'(x_i) \approx a_- u(x_{i-1}) + a_0 u(x_i) + a_+ u(x_{i+1}) =: D_0 u(x_i)$$

- Taylor expansion:

$$u(x_{i-1}) = u(x_i) - h_i u'(x_i) + \frac{1}{2} h_i^2 u''(x_i) + O(h^3)$$
$$u(x_{i+1}) = u(x_i) + h_{i+1} u'(x_i) + \frac{1}{2} h_{i+1}^2 u''(x_i) + O(h^3)$$

Nonuniform Grids (cont)

- Inserting into the expression for $D_0 u$, we obtain after coefficient comparison

$$\begin{aligned} a_- &= \frac{-h_{i+1}}{h_i(h_i + h_{i+1})} \\ a_0 &= \frac{h_{i+1} - h_i}{h_i h_{i+1}} \\ a_+ &= \frac{h_i}{h_{i+1}(h_i + h_{i+1})} \end{aligned}$$

  and

$$D_0 u(x_i) - u'(x_i) = \frac{1}{6} h_i h_{i+1} u'''(x_i) + \dots$$

- For an equidistant grid, the coefficients reduce to $a_- = -1/2h$, $a_0 = 0$, $a_+ = 1/2h$.
- One sided expressions??

An Alternative Idea

- Assume that the grid is created using a mapping $\phi : [0,1] \to [a,b]$ with $x_i = \phi(s_i)$, $i = 0, \ldots, m$ with a uniform grid
$$s_i = i\sigma, \quad \sigma = m^{-1}.$$
- Then, $du/ds = du/dx \cdot dx/ds$, and
$$u_x(x_i) \approx \frac{1}{dx(s_i)/ds} \frac{u_{i+1} - u_{i-1}}{2\sigma}$$
is a second order approximation.

And Another Idea

- If the derivative dx/ds is not known, it can be approximated with second order accuracy by

$$\frac{dx}{ds}(s_i) \approx \frac{x_{i+1} - x_{i-1}}{2\sigma}$$

such that

$$u_x(x_i) \approx \frac{u_{i+1} - u_{i-1}}{x_{i+1} - x_{i-1}}$$

is second order accurate!

- Needed: ϕ is a smooth mapping!
- Note: *We need only two grid points in order to obtain the same order of accuracy as in the approximation in physical domain.*

Approximation of u''

Going either way, we have an approximation

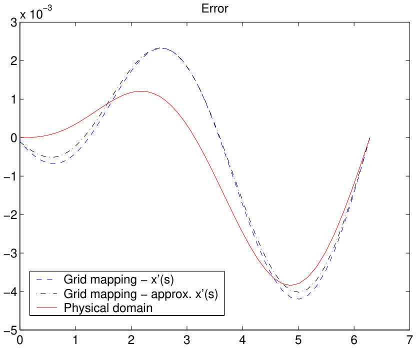
$$u'(x_i) \approx D_0 u_i.$$

A second order approximation to the second derivative can be defined by

$$u''(x_i) \approx D_2 u_i = D_0 D_0 u_i.$$

This approximation evaluates to a five-point stencil!

Example: Comparison of Accuracy



$$u(x) = \sin x$$
$$x(s) = 2\pi \frac{1 + \tanh(\delta(s-1)/2)}{\tanh(\delta/2)}, \quad \delta = 5$$

Hyperbolic tangent stretching, 100 gridpoints.

Conclusions

- All approximations are 2nd order accurate.
- In this simple example, approximation in physical domain is more accurate.
- The stencil (number of grid points used) is larger in physical domain for obtaining the same order of accuracy.

2D: Physical Domain

Ansatz:

$$u_x(x_{i,j}, y_{i,j}) \approx \sum_{k,l} a_{kl} u_{i+k,j+l}$$

Taylor expansion around $(x_{i,j}, y_{i,j})$:

$$\begin{aligned}
&\sum_{k,l} a_{k,l} u_{i+k,j+l} \\
&\quad = \sum_{k,l} a_{k,l} \sum_{\nu=0} \frac{1}{\nu!} \left((x_{i+k,j+l} - x_{i,j})\frac{\partial}{\partial x} + (y_{i+k,j+l} - y_{i,j})\frac{\partial}{\partial y}\right)^{\nu} u \\
&\quad = \sum_{\nu=0} \sum_{p=0}^{\nu} \left[\sum_{k,l} a_{k,l} \frac{1}{\nu!} \binom{\nu}{p} (x_{i+k,j+l} - x_{i,j})^p (y_{i+k,j+l} - y_{i,j})^{\nu-p}\right] \frac{\partial^p}{\partial x^p} \frac{\partial^{\nu-p}}{\partial y^{\nu-p}} u
\end{aligned}$$

$D_{0,x}$ in Physical Domain

First order:

$$\sum_{k,l} a_{k,l} = 0$$

$$\sum_{k,l} a_{k,l}(x_{i+k,j+l} - x_{i,j}) = 1$$

$$\sum_{k,l} a_{k,l}(y_{i+k,j+l} - y_{i,j}) = 0$$

Second order additionally:

$$\sum_{k,l} a_{k,l}(x_{i+k,j+l} - x_{i,j})^2 = 0$$

$$\sum_{k,l} a_{k,l}(x_{i+k,j+l} - x_{i,j})(y_{i+k,j+l} - y_{i,j}) = 0$$

$$\sum_{k,l} a_{k,l}(y_{i+k,j+l} - y_{i,j})^2 = 0$$

So we expect 6 gridpoints necessary for second order accuracy!

Stencil in Reference Coordinates

Remember:

- Let Φ to a (smooth) one-to-one mapping $\Phi : [0,1]^2 \to \Omega$.
- For given m, n, a uniform grid on $[0,1]^2$ can be defined by:

$$\begin{aligned} \xi_i &= ih_1, \quad h_1 = 1/m, \quad i = 0,\dots,m,\\ \eta_j &= jh_2, \quad h_2 = 1/n, \quad j = 0,\dots,n. \end{aligned}$$

- A strucured grid on Ω can then simply be obtained via

$$x_{ij} = \Phi_x(\xi_i,\eta_j), \quad y_{ij} = \Phi_y(\xi_i,\eta_j), \quad i = 0,\dots,m, j = 0,\dots,n.$$

Reference Coordinates (cont)

- Using the chain rule of differentiation, we obtain

$$\frac{\partial u(x,y)}{\partial \xi} = \frac{\partial u}{\partial x}\cdot\frac{\partial \Phi_x}{\partial \xi} + \frac{\partial u}{\partial y}\cdot\frac{\partial \Phi_y}{\partial \xi}$$
$$\frac{\partial u(x,y)}{\partial \eta} = \frac{\partial u}{\partial x}\cdot\frac{\partial \Phi_x}{\partial \eta} + \frac{\partial u}{\partial y}\cdot\frac{\partial \Phi_y}{\partial \eta}$$

  Since the transformation Φ_x, Φ_y is known, this is a linear system for the partial derivatives $\partial u/\partial x, \partial u/\partial y$.

- Let

$$J = \begin{pmatrix} \frac{\partial \Phi_x}{\partial \xi} & \frac{\partial \Phi_y}{\partial \xi} \\ \frac{\partial \Phi_x}{\partial \eta} & \frac{\partial \Phi_y}{\partial \eta} \end{pmatrix}$$

  Then

$$\frac{\partial u}{\partial x} = \frac{1}{\det J}\left(\frac{\partial u}{\partial \xi}\cdot\frac{\partial \Phi_y}{\partial \eta} - \frac{\partial u}{\partial \eta}\cdot\frac{\partial \Phi_y}{\partial \xi}\right)$$
$$\frac{\partial u}{\partial x} = \frac{1}{\det J}\left(\frac{\partial u}{\partial \eta}\cdot\frac{\partial \Phi_x}{\partial \xi} - \frac{\partial u}{\partial \xi}\cdot\frac{\partial \Phi_x}{\partial \eta}\right)$$

Referens Coordinates (cont)

- The derivatives with respect to reference coordinates can be approximated by standard stencils (4-point stencil).
- Once all partial derivatives w r t ξ have been evaluated, the necessary partial derivatives w r t x, y can be computed.

Class for Grid Functions: Requirements

- (Scalar) grid functions are defined on grids.
- We are using structured grids as represented in the class `Domain`.
- Operations allowed with grid functions:
  - Addition, multiplication by a scalar (they form a vector space)
  - Pointwise multiplication (together, this becomes a commutative algebra)
  - Differentiation (e.g., by finite differences)
  - Computation of norms
  - Integration (? maybe)

Further Considerations

- In the two-dimensional case, many of these operations are already implemented in the `Matrix` class!
- However, some operations are not meaningful for grid functions, e.g., matrix-matrix multiplication.
- A grid functions lives only on a specific grid:
  - Shall the grid be part of an object?
  - Many grid functions share the same grid!
  - Algebraic manipulations are only defined for grid functions living on the same grid

Remember: The Matrix Class

```
class Matrix {
  int m, n;  // should be size_t
  double *A;
public:
  Matrix(int m_ = 0, int n_ = 0) : m(m_), n(n_),
            A(nullptr) {
    if (m*n > 0) {
      A = new double[m*n];
      std::fill(A,A+M*n,0.0);
    }
}
// etc
};
```

Implementation of Grid Functions

```
class GFkt {
  private:
    Matrix u;
    Domain *grid;
  public:
    GFkt(Domain *grid_) : u(grid_->xsize()+1,
                           grid_->ysize()+1), grid(grid_) {}
    GFkt(const GFkt& U) : u(U.u), grid(U.grid) {}
    GFkt& opearator=(const GFkt& U);
    GFkt operator+(const GFkt& U) const;
    GFkt operator*(const GFkt& U) const;
// etc
};
```

A Sample Implementation

```
GFkt GFkt::operator+(const GFkt& U) const {
  if (grid == U.grid) {  // defined on the same grid?
     GFkt tmp(grid);
     tmp.u = u+U.u; // Matrix::operator+()
     return tmp;
  }
  else error();
}

GFkt GFkt::operator*(const GFkt& U) const {
  if (grid == U.grid) {  // defined on the same grid?
    GFkt tmp(grid);
    for (int j = 0; j <= grid.ysize(); j++)
      for (int i = 0; i <= grid.xsize(); i++)
        tmp.u(i,j) = u(i,j)*U.u(i,j);
    return tmp;
  }
  else error();
}
```

A Problem And Its Solution

- The grid is handled by the caller.
- In the above implementation, the caller may delete the grid such that all objects referring to it have a dangling pointer!

- In C++ 11 there is a solution: smart pointers
- Smart pointers belong to the C++ library, include file: `memory`

Smart Pointers

- There are two types of them: shared_ptr and unique_ptr.
- Both classes are in fact template classes: The template argument is a typename.
- *shared_ptr uses a reference count: As soon as the reference count reaches 0, the dynamic object will be destroyed. But not earlier!*
- This way, all resources will be freed (including dynamic memory).
- *C-type pointers and smart pointers cannot be mixed!* There is always an explicit type cast necessary! Recommendation: Avoid mixing.

Smart Pointers (cont)

- Create a smart pointer, initialize it to 0 (`nullptr`):

  ```
  shared_ptr<class> p1;
  ```

- The equivalent of new:

  ```
  shared_ptr<class> p2 = make_shared<class>(args);
  ```

  The following statement is in error:

  ```
  shared_ptr<class> p3 = new class(args); // Error!
  ```

  But this works:

  ```
  shared_ptr<class> p3 =
                shared_ptr<class>(new class(args));
  ```

- There is no equivalent of `delete` needed.

A Better Implementation of GFkt

```
class GFkt {
  private:
    Matrix u;
    shared_ptr<Domain>  grid;
  public:
    GFkt(shared_ptr<Domain> grid_) :
              u(grid_->xsize()+1,grid_->ysize()+1),
              grid(grid_) {}
    GFkt(const GFkt& U) : u(U.u), grid(U.grid) {}
// etc
};
```

Notes:

- *We assume silently that, once a grid has been generated, it will never be changed!*
- It is most probably a good idea to use shared pointers in `Domain`, too:

  ```
  shared_ptr<Curvebase> sides[4];
  ```

Implementation of $D_{0,x}$

```
GFkt GFkt::D0x() const {
  GFkt tmp(grid);
  if (grid->grid_valid()) {
    // generate derivative in tmp
    // according to one of the possibilities above
  }
  return tmp;
}
```

- The function `D0y` can be implemented similarly.
- In order to reduce overhead, it might be a good idea to implement even

```
void GFkt::D0xy(GFkt *dx, GFkt *dy) const;
```

Boundary Conditions

Name	Prescribed	Interpretation
Dirichlet	u	Fixed temperature
Neumann	$\partial u/\partial n$	Energy flow
Robin (mixed)	$\partial u/\partial n + f(u)$	Temperature dependent flow
Periodic		

Boundary conditions have a crucial impact on the solution.

What are Boundary Conditions?

1. The mathematician's point of view:

$$\boxed{\begin{array}{l} \quad\text{domain} \\ +\ \text{differential equation} \\ +\ \text{boundary conditions} \end{array}}$$

2. The physicist's point of view:

differential equation	$\longrightarrow$	physics
domain	$\longrightarrow$	space
boundary conditions	$\longrightarrow$	influence of outer world

3. The software engineer's point of view:

differential equation	$\longrightarrow$	expression of differentials
domain	$\longrightarrow$	grid
boundary conditions	$\longrightarrow$	what??

Object-Oriented Representation

- As part of the PDE
  - mathematical interpretation
  - requires high-level representation of equation and discretization
  - difficult to obtain efficiency
- As part of the grid function
  - mathematically correct
  - no class for PDEs needed
  - convenient for exlicit time-stepping
- As part of the operator (e.g., D_0)
  - convenient for implicit and explicit methods
  - can be difficult to implement
  - may encounter mathematical contradictions if used wronly

A First Attempt

Associate boundary conditions with grid functions:

```
class Solution {
  public:
    Solution(Domain *D) : sol(D) {}
    ~Solution();
    void timesteps(double dt, int nsteps);
    void init();  // Set initial condition
    void print();
  private:
    GFkt sol;
    void impose_bc();
};
```

`impose_bc()` will be called in `timesteps()` for imposing the boundary conditions.

Discussion

- The proposed implementation is questionable because the boundary conditions and timestepping are "hardwired".
- It is better to have a *class* for boundary conditions:

```
class BCtype {
  public:
    BCtype(GFkt& u, int boundary_id);
    virtual void impose(GFkt& u) = 0;
};
```

- The actual definition of the boundary condition takes place in derived classes.
- This way, several boundaries can share the same condition (e.g., homogeneous Dirichlet conditions).
- Classes can be derived for Dirichlet, Neumann, Robin boundary conditions.

Example Implementation

Assumptions:

- The grid has four distinct edges (as ours in the previous `Domain` class).
- Each edge is associated with one boundary condition, only.

Then:

```
class Solution {
  public:
    Solution(Domain *D) : sol(D) {}
    ~Solution();
    void print();
  private:
    GFkt sol;
    shared_ptr<BCtype> bcs[4];
    virtual void init() = 0;
    virtual void bc() = 0;
};
```

We have separated: the grid, the equation, the initial conditions, and the boundary conditions.

Time Stepping

For the heat equation in 2D, we can implement the explicit Euler method now:

```
Solution u(&d);
u.init();
for (int step=0; step < maxsteps; step++) {
  u += dt*(u.D2x()+u.D2y());
  t += dt;
  u.bc();
}
```

(Provided the missing functions are implemented along the lines provided before)

Summary

- Finite difference approximations on structured grids.
- Smart pointers
- Implementation strategies for differential operators, boundary conditions, and time steppers.

Course Summary

C++

- Basic elements of C++
- Abstract data types, C++ classes
- Constructors, destructors, memory management, copy, move
- Operator overloading
- Inheritance, abstract classes
- Templates, STL
- I/O

Scientific Computing

- Structured grids, differential operators, boundary conditions
- Implemetation strategies and their C++ tools
- Efficient programming
- Scientific libraries