Matlab – Object-Oriented Programming

Paul Schrimpf

January 14, 2009

Programming Paradigms

- Procedural
  - Do this, then do that ...
  - Most of what we have seen so far
- Functional
  - Operate on functions
  - e.g. LISP, Scheme, Haskell
- Object-oriented
  - Focus on code reuse and reliability
  - An object is data and methods to manipulate it
  - Take components that are used repeatedly and share characteristics and implement as a class
- others ...

Object-oriented Lingo

- A **class** is a data structure and methods that act on it
- An **object** is a specific instance of a class
  - e.g. a double is a class with methods such as `+`, `-`, `*`, `exp()`
- **Encapsulation** refers to the fact that a user of a class should only need to know what a method does, not how it is implemented
  - e.g. Do not need to worry about how doubles are stored or how + works
- A **subclass** is a specialized version of a **parent** class. Subclasses **inherit** data and methods from their parent.
  - e.g. double and int might be subclasses of a generic numeric class

More OO Lingo

- **Abstraction** means writing code that operates at the highest level class possible
  - e.g. most arithmetic operations work with any numeric class
- An **abstraction barrier** refers to the fact that as long as we do not change a given class, changes above it should not require any changes below it, and changes below it should not require any changes above
  - e.g. doing a different sequence of arithmetic operations does not require changing int or double, and changing the implementation of int or double should not require changing a sequence of arithmetic operations

Example: function series class

- Often, we want to approximate an unknown function by a series of functions (as in the dynamic programming example covered earlier)
- Many types of series: orthogonal polynomials, Fourier series, splines, etc
- For any series, we should be able to evaluate at a point, add, subtract, multiply, differentiate, integrate, and construct to give a best approximation to a function
- This suggests writing a generic function series parent class with these methods, and then writing specific types of series as subclasses

Example: function series class

- This code should be in a file named `seriesFn.m` in a directory named `@seriesFn` (In older versions of matlab (before 7.5?) the way of organizing classes and methods was different.)

```
classdef seriesFn
  properties % the data
    order   % order of series
    coeff   % coefficients
    dim     % dimension
    powers  % powers that go with coefficients
  end
  methods (Abstract=true) % these are methods that are only
                          % implemented in child classes
    y = sval(f,x)       % evaluate series
    d = derivative(f)   % create derivative
    F = integral(f,lo,hi) % evaluate integral
    c = mtimes(a,b)     % multiplication
    s = approxFn(s,fn,tol) % make s approximate fn
  end
```

Example: function series class – constructor

- Constructors create a new object
- Child classes can redefine their constructors (or any other method) if they do not, they inherit their parent's constructor
- The following code would go inside a methods block inside classdef seriesFn

```
function f = seriesFn(order,coeff,tol,dim)
  f.order = order;
  f.dim = dim;
  f.powers = intVecsLessThan(f.order,f.dim)';
  if (length(coeff)<size(f.powers,1))
    warning('order=%d, but only %d coeff\n',order, ...
            length(coeff));
  end
  f.coeff = coeff;
  f.coeff(end+1:size(f.powers,2)) = 0;
  f.coeff = reshape(f.coeff,1,numel(f.coeff));
end
```

Example: function series class – operator overloading

- Classes can redefine their own versions of operators (`+`, `-`, `*`, `()`, `:`, etc)
- The following code would go inside a methods block inside classdef seriesFn
- This version of plus will be called whenever someone writes `a + b` and either `a` or `b` is a seriesFn

```
1  function c=plus(a,b)
2    if ¬isa(a,'seriesFn') || ¬isa(b,'seriesFn')
3      error('Both arguments must be seriesFn objects');
4    end
5    order = max(a.order,b.order);
6    c = seriesFn(order,zeros(1,order+1));
7    if (a.order>b.order)
8      c.coeff = a.coeff;
9      c.coeff(1:length(b.coeff)) = c.coeff(1:length(b.coeff)) + b
10   elseif (b.order>a.order)
11     c.coeff = b.coeff;
12     c.coeff(1:length(a.coeff)) = c.coeff(1:length(a.coeff)) + a
13   else
14     c.coeff = a.coeff+b.coeff;
15   end
16 end % function plus
```

Example: function series class – subclass

```
classdef polynomial < seriesFn
  methods
    function c=mtimes(a,b)
      c = polynomial(a.order+b.order, ...
                     conv(a.coeff(end:-1:1),b.coeff(end:-1:1)))
      c.coeff = c.coeff(end:-1:1);
    end
    function y = sval(f,x)
      y = polyval(f.coeff(end:-1:1),x);
    end
    function df = derivative(f)
      df=polynomial(f.order - 1,f.coeff(2:end).*(1:f.order));
    end
    % ... more methods omitted ...
  end
end
```

Forward Automatic Differentiation

- Forward automatic differentiation computes derivatives by applying the chain rule to each operation in a computation
  - e.g. for $x = 2$; $y = x^2$; $z = \log(y)$; forward AD would
    1. $x = 2$, $\partial_x = 1$
    2. $y = x^2 = 4$, $\partial_y = 2x\partial_x = 4$
    3. $z = \log(y)$, $\partial_z = \frac{1}{y}\partial_y = 2$
- We can write a class that stores the value of a number and its derivative, overload every arithmetic operator to work with that class, and then by using this class in place of doubles, we will be able to compute the derivative of any function without changing our code

Example: Forward AD

```
1  classdef autodiff
2    properties
3      val   % value
4      deriv % derivative, deriv(:,:,i) is dval / dx_i
5    end % properties
6    methods
7      function dx = autodiff(x)  % Constructor
8        % ... body omitted ...
9      end % function autodiff
10     % Accessors
11     function v = value(x)
12       v = x.val;
13     end
14     function d = diff(x)
15       d = x.deriv;
16     end
```

Example: Forward AD

```
1       function c=mtimes(a,b)
2         c = autodiff([]);
3         if (isa(a,'autodiff'))
4           if (isa(b,'autodiff'))
5             c.val = a.val*b.val;
6             c.deriv = zeros([size(c.val) size(a.deriv,3)]);
7             for i=1:size(a.deriv,3)
8               c.deriv(:,:,i) = a.deriv(:,:,i) *b.val + ...
9                   a.val*b.deriv(:,:,i);
10            end
11          else
12            c.val = a.val * b;
13            c.deriv = zeros([size(c.val) size(a.deriv,3)]);
14            for i=1:size(a.deriv,3)
15              c.deriv(:,:,i) = a.deriv(:,:,i)*b;
16            end
17          end
18        else
19        % ... other cases omitted ...
20      end % mtimes()
```

Example: Forward AD – usage

- Want to differentiate `fn(x)` with respect to x

```
% x is a double vector or matrix
x = autodiff(x); % make an autodiff object from x
fAD = fn(x); % compute fn and its derivative
fval = value(fAD); % extract the function value
df = diff(fAD); % extract the derivative
```

- For a more complete example, see `binChoiceLikeAD.m`, which uses autodiff on the probit likelihood

Exercises

1. Incorporate objects into the dynamic programming example from earlier. You might begin by making it use the `serisFn` class described above.
2. Add to the autodiff class. It is incomplete. Many methods that work for double matrices have not been implemented. Particularly important and easy methods that need to be implemented include size, ndims, length, and numel. For motivation, you could try making the autodiff class work with some other code that you have written.
3. If the autodiff class was well designed, it would allow doing something like `x = autodiff(autodiff(x))` to compute 2nd and higher order derivatives. Does this work? If not, try to make it work.
4. Design a class hierarchy for datasets. Every econometric program needs to deal with data. A well designed class should make dealing with data easier.
5. Think about the patterns in the type of programs that you most often write, or expect to write. Design classes that will help organize your programs.