

Functional Programming for Numerical Analysis

Judah Jacobson

October 19, 2009

Overview

- A taste of Functional Programming
- It's useful for numerical applications
- Mostly code examples

Why FP

A method of language design/usage which lets you...

- Eliminate repetitive code
- Produce more modular and reuseable code
- Write code which closely matches the equations you're modelling

Examples

- Numerical integration
- Iterative methods
 - Newton's method
 - Conjugate Gradient (i.e., 270C)
- Finite Difference PDE solvers (i.e., 269B)

Avoiding Indices

```
def scaleVector(x,a)
  array b[a.size]
  for(i=0; i<a.size; i++)
    b[i]=x*a[i]
  return b
```

- One solution (OOP): foreach keyword

```
def scaleVector(s,a)
  array b(a.size)
  foreach(i in a.indices)
    b[i]=x*a[i]
  return b
```

- Can we do better?

New thing #1: Higher order functions

- `map` takes a function as the first argument
- `map(f, a)` produces a new array by applying `f` to each element of the array `a`.

```
def double(x)  
  return 2*x
```

```
def square(x)  
  return x*x
```

```
> a = [1,2,3]  
> map(double, a)  
[2,4,6]
```

```
> map(square, [1,3,5,7])  
[1,9,25,49]
```

New thing #2: Anonymous functions

```
> double = x => 2*x
```

```
> double(3)
```

```
6
```

```
> sub42 = t => t-42
```

```
> foo = sub42
```

```
> foo(42)
```

```
0
```

```
> bar = (x,y) => { z = x-15; return x*y }
```

```
> bar(4,9)
```

Anonymous map

```
> double = x => 2*x
```

```
> map(double, [1,2,3])
```

```
[2,4,6]
```

```
> map( x => x*x, [1,3,5,7])
```

```
[1,9,25,49]
```

```
> map( t => t+7, [1,3,5,7])
```

```
[8,10,12,14]
```

Closures

- Anonymous functions can use variables from their environment:

```
> t=5
```

```
> map(x => x+t, [1,2,3])  
[6,7,8]
```

- This leads to our first useful function:

```
def scale(c,a)  
  return map(x => c*x,a)
```

- Note the lack of any explicit loops, indices or array initializations!

Another HOF: fold

- fold combines all the elements of an array into a single argument

```
> add = (x,y) => x+y
```

```
> fold(add, [1,3,5,7]) // ((1+3)+5)+7  
16
```

- Leading to other useful functions:

```
def sum(a)  
  return fold(add, a)
```

```
def product(a)  
  return fold((x,y)=>x*y, a)
```

Norms

```
def norm1(a) // a=[1, -2, -3]
  b = map(abs, a) // b=[1, 2, 3]
  return sum(b) // return 1+2+3=6
```

```
def norm2(a)
  return sqrt(sum(map(x=>x*x, a)))
```

```
def normN(n, a)
  return (sum(map(x=>abs(x)^n, a)))^(1/n)
```

Numerical integration

$$\int_a^b f(x) \approx \Delta x \sum_{i=1}^{n-1} f(x_i)$$

$$\Delta x = \frac{b - a}{n}$$

$$x_i = a + i\Delta x$$

```
> range(5)  
[0, 1, 2, 3, 4]
```

```
def samples(a,b,n)  
    dx = (b - a)/n  
    return map(i => a+i*dx, range(n))
```

```
> samples(0,10,4)  
[0, 2.5, 5, 7.5]
```

$$\int_a^b f(x) \approx \Delta x \sum_{i=1}^{n-1} f(x_i)$$

```
def samples(a,b,n)
  dx = (b-a)/n
  return map(i => a+i*dx, range(n))
```

```
def integrate(f,a,b,n)
  fx = map(f,samples(a,b,n))
  dx = (b-a)/n
  return dx * sum(xs)
```

```
> integrate(sin, 0, pi, 100)
1.9998
```

```
> integrate(x=>2*x+1, 0, 10, 1000)
109.8999
```

Pure functions

- All of the functions we've written so far have no "side effects" such as
 - Modifying global or static variables
 - File I/O, other system calls
- Only observable effect: return a value
- Easier to predict program behavior, e.g.:
 - $f(x)+g(x)$ vs $g(x)+f(x)$
 - index order used by `map(f, x)`

(II) Iterative methods

- Looping is performed in FP using recursion:

```
def gcd(x,y)
  if (y==0) return x
  else
    return gcd(y,x%y)
```

- Compare to the imperative equivalent:

```
def gcd(x,y)
  while(y!=0)
    temp=x
    x=y
    y=temp%y
  return x
```

Tail recursion

- A function is *tail recursive* if the last thing it does is to call itself.

```
def print_squares(k)
  if (k < 0) return
  else
    x = k * k
    print(k + ":" + x + "\n")
    print_squares(k - 1)
```

- FP languages optimize tail-recursive functions into flat loops

Newton's Method

- Approximate $\text{sqrt}(z)$: $x_{i+1} = x_i/2 + z/(2x_i)$

```
def newton_sqrt(z, tol, x)
  if (abs(x*x-z)<tol)
    return x
  else return
    newton_sqrt(z, tol, x/2+z/(2x))
```

```
> newton_sqrt(16, 0.0001, 8)
4.000000185844589
```

Avoid explicit recursion

- `iterateN` repeatedly calls `f` on its input:

```
def iterateN(n, f, x)
  if (n <= 0) return x
  else iterateN(n-1, f, f(x))
```

```
> iterateN(10, x => 2*x, 1)
1024
```

- Which lets us write `newtonSqrt` without recursion:

```
def newtonSqrt(z, n)
  let advance = x => x/2 + z/(2*x)
  iterateUntil(n, advance, z/2)
```

Stopping condition

```
def iterateUntil(stop, f, x)
  if (stop(x)) return x
  else iterateUntil(stop, f, f(x))

> iterateUntil(x=>x>10, x=>x+2, 1)
11

def newtonSqrt(z, tol)
  let stop = x => abs(x*x-z)<tol
  let advance = x => x/2 + z/(2*x)
  iterateUntil(stop, advance, z/2)
```

Conjugate Gradient

Solve $Ax=b$ for symmetric, positive definite A

Initial conditions:

$$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$$

$$\mathbf{p}_0 = \mathbf{r}_0$$

$$k = 0$$

Loop:

$$\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k$$

If $\|\mathbf{r}_{k+1}\| < \epsilon$ then return \mathbf{x}_{k+1} .

$$\beta_k = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$$

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$$

$$k = k + 1$$

Imperative version

```
def conjgrad(A,b,x0,tol)
    x=x0
    r=b-A*x
    p=r
    while(true)
        w=A*p
         $\alpha=(r' * r)/(p' * w)$ 
        x=x+ $\alpha$ *p
        r1=r- $\alpha$ *w
        if ( norm2(r1)<tol ) break
         $\beta=(r1' * r1)/(r' * r)$ 
        r=r1
        p=r+ $\beta$ *p
    return x
```

Recursive version

```
def conjgrad(A,b,x0,tol)
  conjgrad_loop = (x,r,p) => {
    w=A*p
     $\alpha=(r' * r)/(p' * w)$ 
    x1=x+ $\alpha$ *p
    r1=r- $\alpha$ *w
    if (norm2(r1)<tol) return x
    else
       $\beta=(r1' * r1)/(r' * r)$ 
      return conjgrad_loop(x1,r1,r1+ $\beta$ *p)
  }
  r0=b-A*x0
  return conjgrad_loop(x0,r0,r0)
```

Sparse CG

```
def sparseApply(B,x) ...
```

```
def conjgrad(Af,b,x0,tol)
  def conjgrad_loop(x,r,p)
    w=Af(p)
    ...
```

```
> A=... // dense
```

```
> conjgrad(x=>A*x,b,x0,tol)
```

```
> B=... // sparse version of A
```

```
> conjgrad(x=>sparseApply(B,x), b, x0, tol)
```

- But if we can pass in any function we want...
- Let the function *itself* be the representation!

```
def apply1DLaplacian(x)
    y=new array(x.size)
    y[0] = 2*x[0] - x[1]
    y[n] = 2*x[n] - x[n-1]
    for (i=1; i<x.size-1; i++)
        y[i]=2*x[i] - x[i-1] - x[i+1]
    return y
```

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

> ...

> conjgrad(apply1DLaplacian, b, x0, tol)

Testing

- It's easy to test that function separately:

```
> sins = map(sin, samples(0, pi, 100))  
> h = pi / (100 - 1)  
> norm2( sins - (1 / (h^2))  
          * apply1DLaplacian(sins) )
```

(III) Finite Difference Schemes

$v_t = \mathcal{D}(v)$ where $\mathcal{D}(v) = v_{xx}, -v_x, \text{ etc.}$

- Method of Lines: replace the spatial differential operator \mathcal{D} with a discrete approximation \mathcal{D}_h for example:

$$\mathcal{D}(v) = v_{xx}, \mathcal{D}_h(v) = \frac{v_{i-1} - 2v_i + v_{i+1}}{h^2}$$

- Then, discretize in time using an explicit scheme:

$$\frac{v_i^{j+1} - v_i^j}{k} = \frac{v_{i-1}^j - 2v_i^j + v_{i+1}^j}{h^2}$$

Parameters

```
// time sampling
params.n, params.startT, params.endT
// space sampling
params.m, params.startX, params.endX
// initial values
params.initial // this field is a function!

> p.n=100
> ...
> p.initial = x => 3+cos(x)
> v0 = map(p.initial,
           sample(p.startX,p.endX, p.m))
```

Forward Euler

```
def forwardEuler(D,p)
    v0 = map(p.initial,
            sample(p.startX,p.endX, p.m))
    k = (p.endT - p.startT) / p.n
    h = (p.endX - p.startX) / p.m
    return iterateN(p.n, v => v+k*D(v,h), v0)
```

$$v^{j+1} = v^j + k\mathcal{D}_h(v^j)$$

```
// solving v_t=v_xx
> laplace0p = (v,h) =>
    (-1)*apply1DLaplacian(v)/(h^2)
> vn = forwardEuler(laplace0p, params)
```

Other diff ops

- It's easy to use that code for other problems:

```
// approximate v_x by (v_{i+1}-v_i)/h
```

```
def forwardD(v,h)
```

```
    w = array(v.size)
```

```
    w[0]=v[0]-v[v.size-1]
```

```
    for (i=1; i<v.size; i++)
```

```
        w[i]=(v[i+1] - v[i])/h
```

```
    return w
```

```
// solve v_t=v_x
```

```
> vn = forwardEuler(forwardD, params)
```

Higher-order methods

- Generalize by passing the time-differencing operator as another function parameter:

```
def explicitDiff(g,D,p)
  v0 = map(p.initial,
           sample(p.startX,p.endX, p.m))
  k = (p.endT - p.startT) / p.n
  h = (p.endX - p.startX)/p.m
  return iterateN(p.n, v => g(D,v,k), v0)
```

> forwardEulerOp = (D,v,k) => v+k*D(v)

> explicitDiff(forwardEulerOp, forwardD, p)

Runge-Kutta

$RK2(D, v, k) \Rightarrow v + k * D(v + k/2 * D(v))$

```
def RK4(D, v, k)
```

```
    y1=D(v)
```

```
    y2=D(v+0.5*k*v1)
```

```
    y3=D(v+0.5*k*v2)
```

```
    y4=D(v+k*v3)
```

```
    return v + (k/6)*(y1+y2+y3+y4)
```

> explicitDiff(RK2, forwardD, params)

> explicitDiff(RK4, laplace0p, params)

Looking back: what is FP?

A philosophy of language design and use, favoring:

- Higher-order functions: functions-as-data
- Anonymous functions
- Side-effect-free functions
- Recursion over iteration
- Building a program from small reusable functions

Mainstream FP support

- Python: `square = lambda x:x**2`
- Matlab: `square = @(x)(x^2)`
- JavaScript: `function(x){return x^2;}`
- C++ (Boost.Lambda):
`for_each(v.begin(), v.end(), _1 = _1^2);`
- Also C#, Ruby, Perl, ...

FP Languages

- LISP/Scheme: dynamically typed, support metaprogramming
- ML/Haskell: statically typed (usefully)
- Erlang: dynamically typed, very good concurrency support
- Scala (JVM), F# (CLR)

Haskell

- Originally a research language, but has developed into a tool for practical use
- (Some) distinguishing features:
 - Static typing
 - Purity by default; side effects are explicit
- Compiler (GHC) can produce fast code (2-10x gcc depending on skill)
- Good parallelism/concurrency support

Parallelizing in Haskell

- Many consumer PCs are multicore
- `map` creates the new array sequentially
- Haskell provides a parallel version:
`b = parmap(f, a)`
- Each CPU core computes a different section of the elements of `b`
- Usually an automatic speedup! ($\sim 1.7x$ on dual-core)

Purity prevents races

- Problem: race conditions in `b=parmap(f, a)`?
- Haskell requires that `f` be pure, so that the order in which the values of `b` are computed doesn't matter.
- Most of the algorithms we've talked about can be sped up this way
- I also use this for sound processing

Aside: Google MapReduce

- Distributed computing on large clusters
- Turn a task into two steps: "map" + "reduce"
- "map": small subproblem to be performed on each node
- "reduce": combine subproblems => final result
- Example: URL access statistics
 - map: parse a logfile and output # for each URL
 - reduce: for one URL, add up totals from all logs

MapReduce, con't

- Programmer writes a "map" and "reduce" task in a special language (~20 lines)
- Uses C++ or Python to call the MapReduce library
- Automatically distributes and manages jobs between machines
- As before, purity is key -- the MapReduce language doesn't permit side effects.
- Allows great flexibility in splitting up and restarting jobs

Takeaways

- FP applies well to many numerical methods
- The FP style is fun and useful
- Your language probably already supports FP