Charlie Nusbaum
Econ 204B
February 5, 2018

Note on Optimization and Interpolation

Interpolation: There are many different interpolation schemes that one could use. Each of these schemes comes with its own pros and cons. For example, linear interpolation is simple and works well for most problems. Unfortunately, approximating a function using linear interpolation implies that the $f'(x)$ is discontinuous and undefined at the interpolation nodes. Piecewise-cubic interpolation on the other hand ensures that $f'(x)$ is continuous and defined over the interpolation grid, however, it results in small "wiggles" around the true function. Chebyshev appoximation is by far the most popular interpolation scheme as it chooses the interpolation nodes so as to minimize the approximation error. Moreover, the Chebyshev polynomials don't "waste" information in the sense that they are orthogonal. Since this is not a numerical methods class, I won't get into the details. If you'd like to know more, stop by my office.

Below I've inlcuded code for linear interpolation. The first snippet returns a function whereas the second snippet returns the approximation evaluated at a point. Note that the second function was written to be jit-compatible so you should either import jit from numba or simply delete the jit decorator.

```
#########################################################################
# This function performs a linear spline for 1D interpolation and returns an interp

# x := the x-values over which to interpolate (assumes x is ascending order)
# y := the y-values that need interpolating
# a := the value at which to interpolate

def interp_linear(x,y):

    def f1(a):

        if a > x[-1,]:
            val = y[-1,]+(y[-1,]-y[-2,])/(x[-1,]-x[-2,])*(a-x[-1,])

        elif a < x[0,]:
            val = y[0,]+(y[1,]-y[0,])/(x[1,]-x[0,])*(a-x[0,])

        else:
            ind = np.argmax(x>a)
            val = y[ind-1,]+(y[ind,]-y[ind-1,])/(x[ind,]-x[ind-1,])*(a-x[ind-1,])

        return val

    return f1
```

```
##########################################################################
# This function performs a linear spline for 1D interpolation and returns an interp

# I changed this so that it returns the value at a point rather than a function tha
@jit(nopython=True)
def interp_linear_jit(x,y,point):
    n = len(x)
    for i in range(n-1):
        if point >= x[i] and point <= x[i+1]:
            val = y[i] + ((y[i+1]-y[i])/(x[i+1]-x[i]))*(point-x[i])
            break

    return val
```

Optimization: Optimization is a key ingredient to dynamic program. In particular, we need to optimize the relevant value function at each iteration. Here, there are also several approaches that one could take. The first is Golden-Section search. This method is the equivalent of the bisection method, but for optimization. As such, it is quite robust, however, also quite slow. Newton's method on the other hand, is extremely fast, but incredibly sensitive to infection points. Brent's method can be thought of as a half-way between Newton's method and Golden-Section search. Again, this is not a numerical methods class so I will not get into the details. If you'd like to learn more, either come by my office or use any of the abundance of resources that can be found online. Below is a snippet of code to perform Golden-Section search:

```
##########################################################################################

# This function finds the minimum of a function using the golden section method
# fun := The function to optimize
# a   := The first bound on the triple
# b   := The interior point of the triple
# tol := The error tolerance for the stopping criterion => Numerical Recipes sugges
# tolerance of the function values
# maxsteps := The maximum number of iterations
# true := The true optimum if known.... if true=0 then error simply traces out the

def opt_goldensection(fun,a,b,tol=10**(-7),maxsteps=10**3):

    step = 0
    r    = 2/(1+5**(0.5))
    x1   = float(b-r*(b-a))
    x2   = float(a+r*(b-a))
    f1   = fun(x1)
    f2   = fun(x2)

    while step<maxsteps:
        step = step+1
        if f2>f1:
            b  = x2
            x2 = x1
            x1 = b-r*(b-a)
            f2 = f1
            f1 = fun(x1)

        else:
            a  = x1
            x1 = x2
            x2 = a+r*(b-a)
            f1 = f2
            f2 = fun(x2)

        diff = b-a
        if step==maxsteps and maxsteps==10**3:
```

```
            print('No root was found after the maximum number of steps using golden

        elif step == maxsteps:

            print(' ')
            print('The root was not found within tolerance in the max number of ste
            print(' ')

        if abs(diff)<tol:
            break

    x   = 0.5*(a+b)
    val = fun(x)

    return x,val
```