hw09-notes

Dan White

September 30, 2014

1 Notes on guess-and-iterate

Sometimes, the way you setup the two equations to iterate makes the difference between finding a solution or getting incoherent results in a few steps.

As an example of this phenomenon, consider question 2 from hw08. This is a topologically similar circuit to the one of this assignment but with different numbers and equations for the diode. The concepts shown and discussed below are the same, however.

First step is to construct the two mutually-coupled equations with which to iterate. How about the given equation for the diode current as a function of diode voltage:

$i_x = 10^{-12} \left[\exp(v_D/V_T) - 1 \right]$

Then v_D as a function of i_x using Ohm's law and KVL with the resistor:

$v_D = 3.3 - 500 \cdot i_x$

Set these up as functions:

In [1]:
```
# id equation from the diode
def ix_d(vd):
    return 1e-12 * (exp(vd/26e-3) - 1)

# vd equation from the resistor
def vd_r(ix):
    return 3.3 - 500*ix
```

The below function accepts two functions and an initial guess. It is the python code version of our guess-and-iterate procedure.

In [2]:
```
def iterate(func1, func2, guess,
            tol=1e-5,
            verbose=True,
            maxiter=100):
    """Iterates two mutually-coupled equations starting from a first
    guess of y=func1(guess), then x=func2(y), then y=func1(x),
    and so on until:
        a) the relative change in is less than tol
        b) the maximum number of iterations is reached
    """

    i = 0

    #function evaluation wrapper to print its results at each step
    def eval(f, x):
        #print the failing step before raising the original exception
        try:
            y = f(x)
        except:
            if verbose:
                print '%i: *FAIL* = %s( %g )' % (i, f.__name__, x)
            raise

        if verbose: print '%i: %g = %s( %g )' % (i, y, f.__name__, x)
        return y

    #manually do the first step so we have two values of each variable
    x_last = guess
    y_last = eval(func1, guess)
    i += 1

    x = eval(func2, y_last)
    y = eval(func1, x)
    i += 1

    while True:
        y = eval(func2, x)
        x = eval(func1, y)

        if ((abs(y_last - y)/y) < tol and
            (abs(x_last - x)/x) < tol):
            # a) convergence
            return (x, y)
        elif i >= maxiter:
            # b) too many iterationss
            print '*** Reached maximum number of iterations (%i) ***' % i
            return (x, y)
        else:
            #housekeeping, keep going
            x_last = x
            y_last = y
            i += 1
```

Now, pass in our two functions and an initial guess for v_D. How about a guess halfway between the voltage source, 1.65 V.

```
iterate(ix_d, vd_r, 1.65)
```

In [3]:
```
0: 3.63912e+15 = ix_d( 1.65 )
1: -1.81956e+18 = vd_r( 3.63912e+15 )
1: -1e-12 = ix_d( -1.81956e+18 )
2: 9.09779e+20 = vd_r( -1.81956e+18 )
2: inf = ix_d( 9.09779e+20 )
3: -inf = vd_r( inf )
3: -1e-12 = ix_d( -inf )
4: 3.3 = vd_r( -1e-12 )
4: 1.32432e+43 = ix_d( 3.3 )
5: -6.62158e+45 = vd_r( 1.32432e+43 )
5: -1e-12 = ix_d( -6.62158e+45 )
-c:3: RuntimeWarning: overflow encountered in exp
-c:40: RuntimeWarning: invalid value encountered in double_scalars
(-9.9999999999999998e-13, -6.6215848198684692e+45)
```

Out [3]:

!Wow! Do you see what happened? A guess of $v_D = 1.65V$ gave an i_x of $3.6 \times 10^{15} V$, or 3.6 petavolts. From there, the situation just got worse and the iteration didn't nicely converge to anything useful. In fact, the computer stopped the iteration after it started computing bogus numbers.

This code version of guess-and-iterate did not include the most important aspect: ***YOU!*** You would have stopped around step 0 or 1 because the numbers were not working out.

What is missing is a sanity check. From inspecting the circuit, it is clear that v_D can only be in the range of 0 to 3.3 V and the current i_x can only be in the range 0 to (3.3V / 500 Ohm) = 6.6 mA. Add a range check to our `iterate` function to include valid ranges on the variables.

In [4]:

```
def iterate(func1, func2, guess, xlim, ylim,
            tol=1e-6,
            verbose=True,
            maxiter=100):
    """Iterates two mutually-coupled equations starting from a first
    guess of y=func1(guess), then x=func2(y), then y=func1(x),
    and so on until:
        a) the relative change is less than tol
        b) the maximum number of iterations is reached
        c) one of our values goes out of range
    """

    i = 0
    iterations = []

    #function evaluation wrapper to print its results at each step
    def eval(f, x, limits):
        if (x < min(limits) or (x > max(limits))):
            if verbose:
                print '%i: *FAIL* = %s( %g ) *** out of range (%g, %g) ***' % (
                        i, f.__name__, x, limits[0], limits[1])
            raise StopIteration

        #print the failing step before raising the original exception
        # the limit check should catch the normal errors
        try:
            y = f(x)
        except:
            if verbose:
                print '%i: *FAIL* = %s( %g )' % (i, f.__name__, x)
            raise

        if verbose:
            print '%i: %g = %s( %g )' % (i, y, f.__name__, x)

        if len(iterations) % 2 == 0:
            iterations.append((x, y))
        else:
            iterations.append((y, x))

        return y

    try:
        #manually do the first step so we have two values of each variable
        x_last = guess
        x = x_last
        y_last = eval(func1, guess, xlim)
        y = y_last
        x = eval(func2, y_last, ylim)
        i += 1

        while True:
            y = eval(func1, x, xlim)
            x = eval(func2, y, ylim)
            i += 1
            if ((abs(y_last - y)/y) < tol and
                (abs(x_last - x)/x) < tol):
                # a) convergence
                print '*** converged ***'
                raise StopIteration
            elif i >= maxiter:
                # b) too many iterationss
                print '*** Reached maximum number of iterations (%i) ***' % i
                raise StopIteration
            else:
                #housekeeping, keep going
                x_last = x
                y_last = y

    except StopIteration:
        pass

    return x, y, iterations
```

Ok, now setup the iteration again with the same guess but now include the limits on v_D and i_x.

In [5]:
```
ix, vd, s = iterate(ix_d, vd_r, 1.65, [0, 3.3], [0, 6.6e-3])
```

```
0: 3.63912e+15 = ix_d( 1.65 )
0: *FAIL* = vd_r( 3.63912e+15 ) *** out of range (0, 0.0066) ***
```

At least we now have a stated reason for failing. What about a different guess, like $v_D = 0$?

In [6]:
```
ix, vd, s = iterate(ix_d, vd_r, 0, [0, 3.3], [0, 6.6e-3])
```

```
0: 0 = ix_d( 0 )
0: 3.3 = vd_r( 0 )
1: 1.32432e+43 = ix_d( 3.3 )
1: *FAIL* = vd_r( 1.32432e+43 ) *** out of range (0, 0.0066) ***
```

It turns out that, unless we guess exactly right, this will always happen with this set of equations. The points (v_d, i_x) spiral "out of control" as the iteration progresses.

What to do?

How about invert the two equations. Solve each equation the other way around:

Find the current from Ohms law:

$$i_x = \frac{3.3 - v_D}{500}$$

Solve the diode equation for v_D:

$$v_D = V_T \ln\left[1 + \frac{i_x}{I_S}\right]$$

In [7]:
```
def ix_r(vd):
    return (3.3 - vd)/500

def vd_d(ix):
    return 26e-3 * log(1 + ix/1e-12)
```

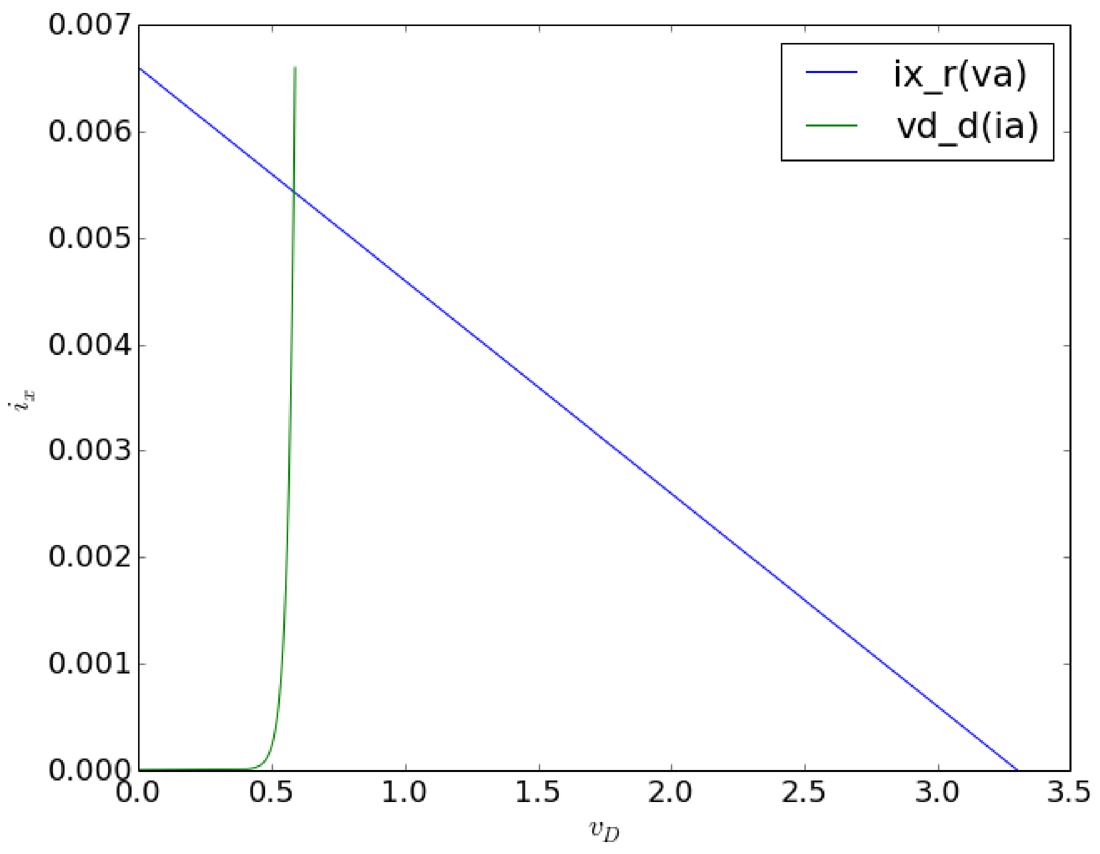
Let's plot the iteration steps as they happen. First plot the two functions themselves on the same plot. Use the same axes, so the v_D plot is mirrored.

In [8]:
```
%config InlineBackend.close_figures = False
plt.close('all')

vd = linspace(0, 3.3, 1e3)
ix = linspace(0, 6.6e-3, 1e3)

plot(vd, ix_r(vd), label='ix_r(va)')
plot(vd_d(ix), ix, label='vd_d(ia)')
xlabel('$v_D$'); ylabel('$i_x$'); legend(loc='best')
```

Out [8]:
```
<matplotlib.legend.Legend at 0x2d9e310>
```



Run the guess-and-iterate again. Use the same initial guess $v_D = 1.65V$.

In [9]:
```
ix, vd, s = iterate(ix_r, vd_d, 1.65, [0, 3.3], [0, 6.6e-3])
```

```
0: 0.0033 = ix_r( 1.65 )
0: 0.569847 = vd_d( 0.0033 )
1: 0.00546031 = ix_r( 0.569847 )
1: 0.58294 = vd_d( 0.00546031 )
2: 0.00543412 = ix_r( 0.58294 )
2: 0.582815 = vd_d( 0.00543412 )
3: 0.00543437 = ix_r( 0.582815 )
3: 0.582816 = vd_d( 0.00543437 )
4: 0.00543437 = ix_r( 0.582816 )
4: 0.582816 = vd_d( 0.00543437 )
*** converged ***
```

Success! We declared convergence after the 4th step.

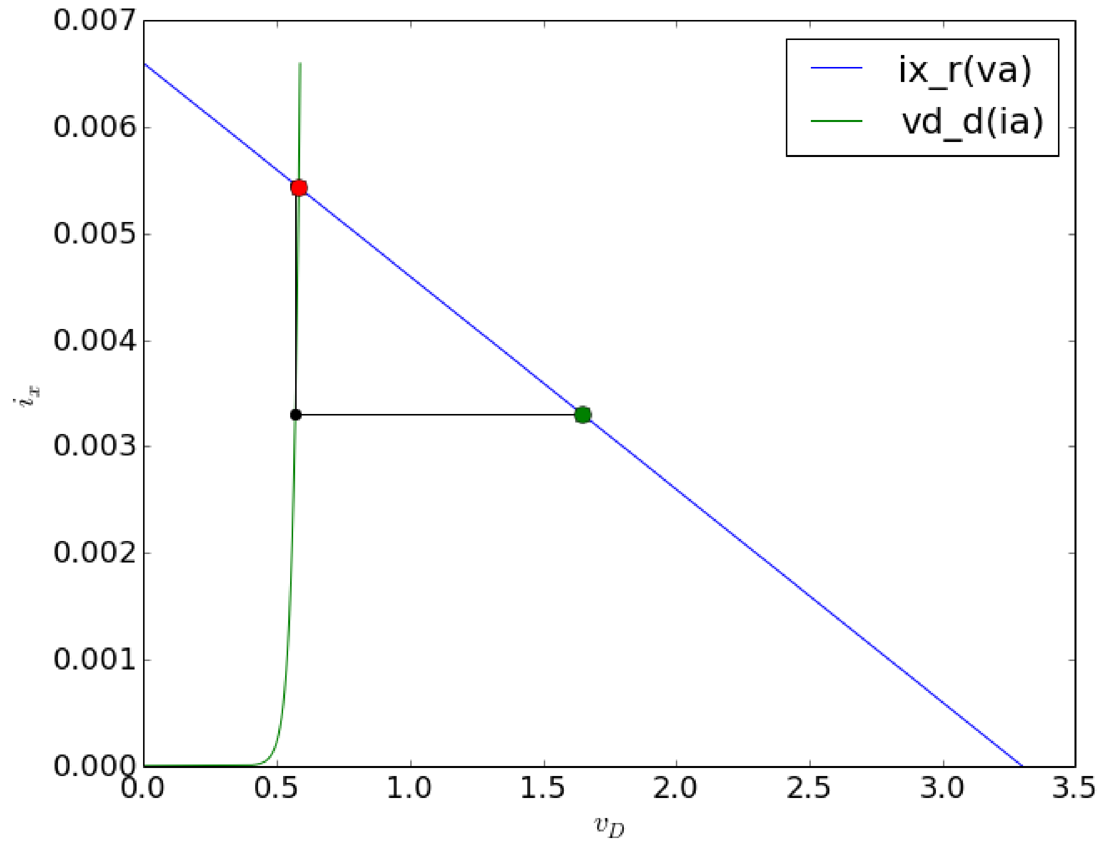
To see what's going on a little better, plot the steps of the iteration on our plot of the two functions from before.

In [10]:
```
# draw lines between the points to visualize the convergence
for i in range(1, len(s)):
    plot([s[i-1][0], s[i][0]],
         [s[i-1][1], s[i][1]], '-ok')

#mark the start and ending points in appropriate colors
plot(s[0][0], s[0][1], 'og', markersize=10)
plot(s[-1][0], s[-1][1], 'or', markersize=10)
```

```
[<matplotlib.lines.Line2D at 0x3359890>]
```

Out [10]:

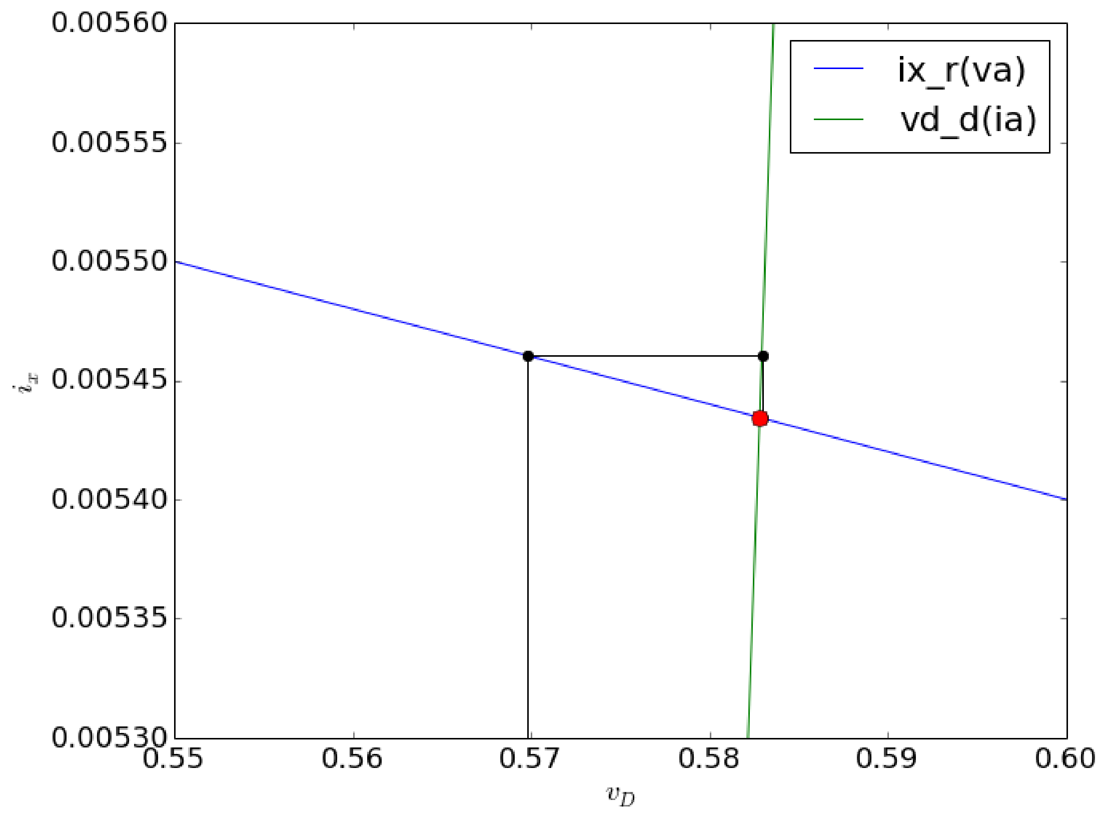


Zoom into near the final answer:

In [11]:
```
ylim((5.3e-3, 5.6e-3))
xlim((0.55, 0.60))
```

```
(0.55, 0.6)
```

Out [11]:



1.1 Summary

Usually, one way of writing the two mutually-coupled equations works fine while the other never works for any initial guess.

Make note: The one that worked had the diode equation using the $\ln()$. This same effect is occurs with the vacuum diode equation of `hw09`. Use the equation version solved which has the $X^{2/3}$ term instead of the $Y^{3/2}$ term. One function has increasing slope as the input increases, the other “flattens out” as the input gets larger.

Kind of makes you want to take a few more math courses like “Numerical Analysis” or “Complex Variables,” doesn’t it?