Solving Hydrogen Atom with Python

We want to write an algorithm that compute energies and charges of the bound states in central potential of the nucleus with charge Z. We will first solve the problem for a single electron (no Coulomb interaction).

The implementation will follow these steps

1. call SciPy routine

   ```
   integrate.odeint
   ```

   to integrate the one-electron Schroedinger equation
   $$-u^{''}(r) + \left(\frac{l(l+1)}{r^2} - \frac{2Z}{r} \right) u(r) = \varepsilon u(r).$$
   Here $\psi_{lm}(\vec{r}) = \frac{u(r)}{r} Y_{lm}(\hat{r})$, distance is measured in units of bohr radius and energy units is Rydberg ($1Ry = 13.6058...eV$)

2. The boundary conditions are $u(0) = 0$ and $u(\infty) = 0$. Use shooting method to obtain wave functions:
   A. Use logarithmic mesh of radial points for integration. Start integrating from a large distance ($R_{max} \sim 100$). At R_{max} choose $u = 0$ and some nonzero (not too large) derivative.
   B. Integrate the Schroedinger equation down to $r = 0$. If your choice for the energy ε corresponds to the bound state, the wave function at $u(r = 0)$ will be zero.

3. Start searching for the first bound state at sufficiently negative energy (for example $\sim -1.2Z^2$) and increase energy in sufficiently small steps to bracket all necessary bound states. Ones the wave function at $r = 0$ changes sign, use root finding routine, for example

   ```
   optimize.brentq,
   ```

   to compute zero to very high precision. Store the index and the energy of the bound state for further processing.
4. Ones bound state energies are found, recompute $u(r)$ for all bound states. Normalize $u(r)$ and plot them.

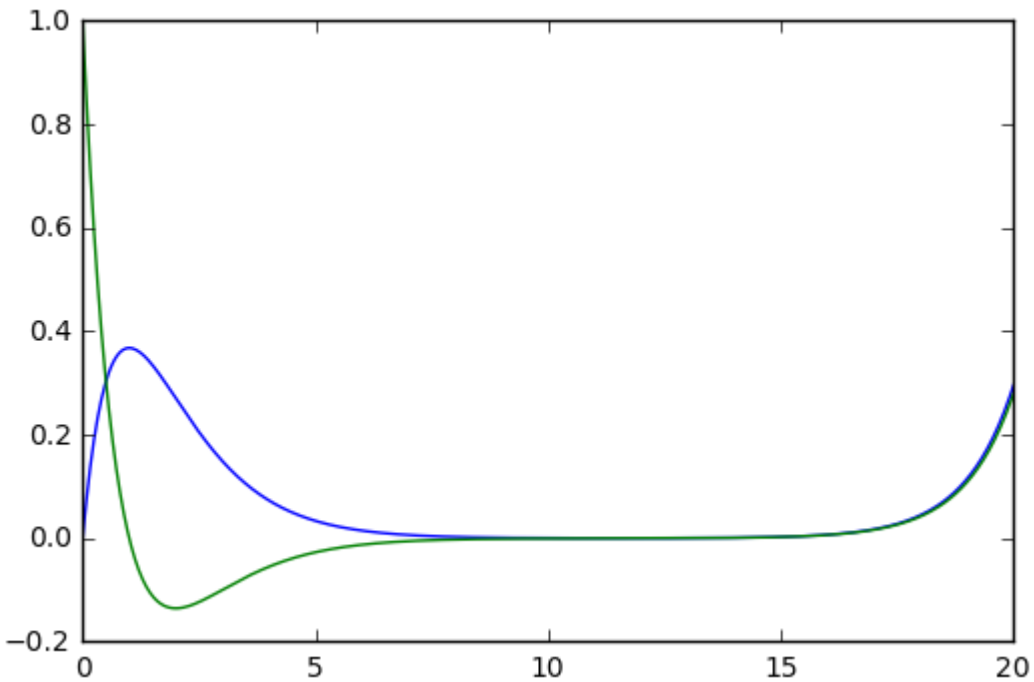
5. Compute electron density for various atoms (for example He, Li, ..) neglecting Coulomb repulsion: Populate first Z lowest laying electron states and compute $\rho = \sum_{lm \in occupied} u_{lm}^2(r)/(4\pi r^2)$. Each state with quantum number l can take $2(2l+1)$ electrons. Be carefull, if atom is not one of the Nobel gases (He, Ne, ...) the last orbital is only partially filled.

In [1]:
```
from scipy import *
from scipy import integrate
from scipy import optimize

def Schroed_deriv(y,r,l,En):
    "Given y=[u,u'] returns dy/dr=[u',u''] "
    (u,up) = y
    return array([up, (l*(l+1)/r**2-2/r-En)*u])
```

First we try linear mesh and forward integration. It is supposed to be unstable. We know the ground state has energy $E_0 = -1Ry$ and we should get $1s$ state with integrating Scroedinger equation.

In [2]:
```
R = linspace(1e-10,20,500)
l=0
E0=-1.0

ur = integrate.odeint(Schroed_deriv, [0.0, 1.0], R, args=(l,E0))
```

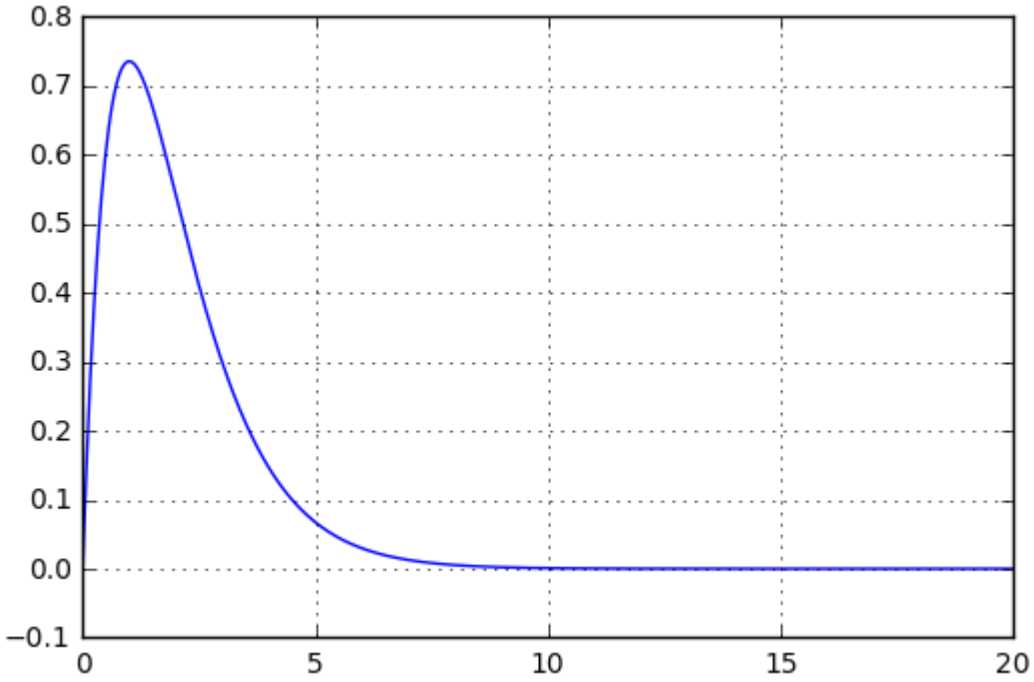
In [3]:
```
from pylab import *
%matplotlib inline

plot(R,ur)
show()
```

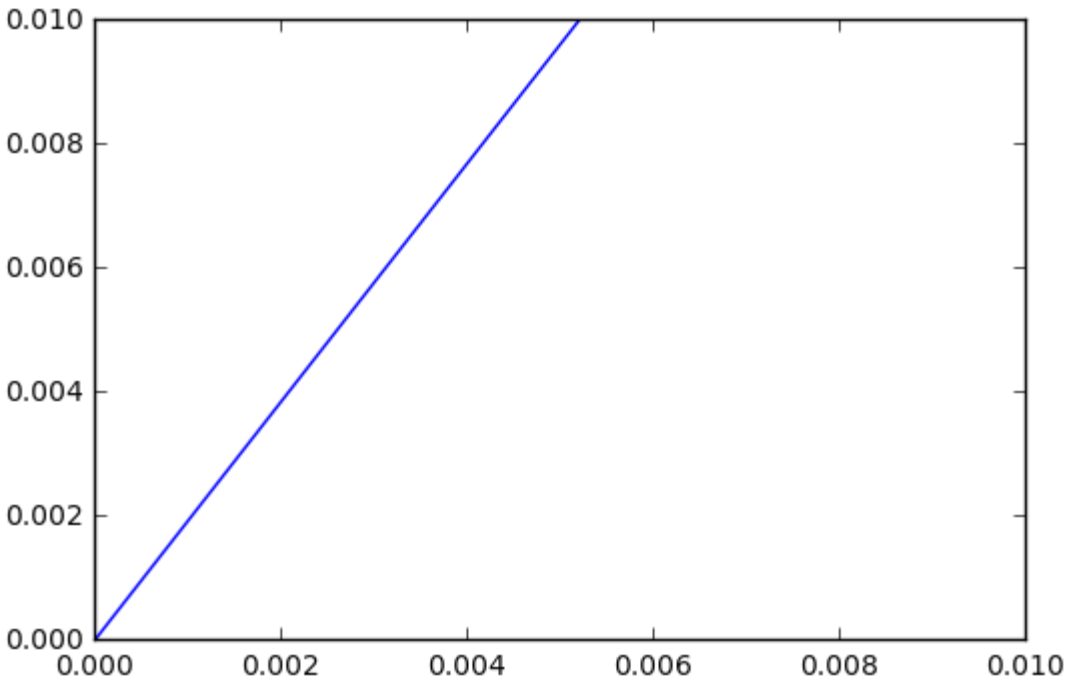


Indeed the integration is unstable, and needs to be done in opposite direction. Let's try from large R.

In [4]:
```
R = linspace(1e-10,20,500)
l=0
E0=-1.0
Rb=R[::-1] # invert the mesh

urb = integrate.odeint(Schroed_deriv, [0.0, -1e-5], Rb, args=(l,E0))
ur = urb[:,0][::-1] # we take u(r) and invert it in R.

norm=integrate.simps(ur**2,x=R)
ur *= 1./sqrt(norm)
```

In [5]:
```
plot(R,ur)
grid()
show()
plot(R,ur)
xlim(0,0.01)
ylim(0,0.01)
show()
```





Clearly the integration from infinity is stable, and we will use it here.

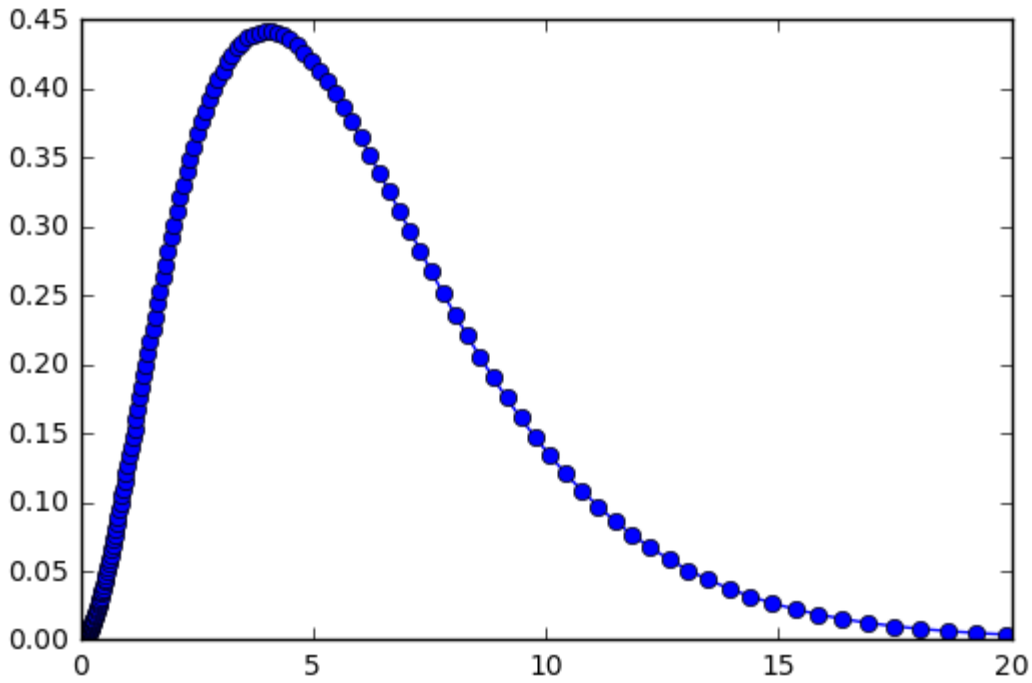
Logarithmic mesh is better suited for higher excited states, as they extend far away.

Lets create a subroutine of what we learned up to now:

In [6]:
```
def SolveSchroedinger(En,l,R):
    Rb=R[::-1]
    du0=-1e-5
    urb=integrate.odeint(Schroed_deriv, [0.0,du0], Rb, args=(l,En))
    ur=urb[:,0][::-1]
    norm=integrate.simps(ur**2,x=R)
    ur *= 1./sqrt(norm)
    return ur
```

In [7]:
```
l=1
En=-1./(2**2)  # 2p orbital

l=1
En = -0.25

Ri = linspace(1e-6,20,500)    # linear mesh already fails for this case
ui = SolveSchroedinger(En,l,Ri)


R = logspace(-5,2.,500)
ur = SolveSchroedinger(En,l,R)


#ylim([-0.1,0.1])
plot(R,ur,'o-')
#plot(Ri,ui,'s-')
xlim([0,20])
```

Out[7]: (0, 20)



Next we create a shooting routine.

In [8]:
```
def Shoot(En,R,l):
    Rb=R[::-1]
    du0=-1e-5
    ub=integrate.odeint(Schroed_deriv, [0.0,du0], Rb, args=(l,En))
    ur=ub[:,0][::-1]
    norm=integrate.simps(ur**2,x=R)
    ur *= 1./sqrt(norm)

    ur = ur/R**l

    f0 = ur[0]
    f1 = ur[1]
    f_at_0 = f0 + (f1-f0)*(0.0-R[0])/(R[1]-R[0])
    return f_at_0
```

In [9]:
```
R = logspace(-5,2.2,500)
Shoot(-1./2**2,R,1)
```

Out[9]:
```
144.55590022465006
```

In [10]:
```
def FindBoundStates(R,l,nmax,Esearch):
    n=0
    Ebnd=[]
    u0 = Shoot(Esearch[0],R,l)
    for i in range(1,len(Esearch)):
        u1 = Shoot(Esearch[i],R,l)
        if u0*u1<0:
            Ebound = optimize.brentq(Shoot,Esearch[i-1],Esearch[i],xtol
=1e-16,args=(R,l))
            Ebnd.append((l,Ebound))
            if len(Ebnd)>nmax: break
            n+=1
            print 'Found bound state at E=%14.9f E_exact=%14.9f l=%d' %
(Ebound, -1.0/(n+l)**2,l)
        u0=u1

    return Ebnd
```

In [11]:
```
Esearch = -1.2/arange(1,20,0.2)**2

R = logspace(-6,2.2,500)

nmax=7
Bnd=[]
for l in range(nmax-1):
    Bnd += FindBoundStates(R,l,nmax-l,Esearch)
```

```
Found bound state at E=  -1.000000014 E_exact=  -1.000000000 l=0
Found bound state at E=  -0.249999998 E_exact=  -0.250000000 l=0
Found bound state at E=  -0.111111111 E_exact=  -0.111111111 l=0
Found bound state at E=  -0.062500001 E_exact=  -0.062500000 l=0
Found bound state at E=  -0.040000000 E_exact=  -0.040000000 l=0
Found bound state at E=  -0.027777780 E_exact=  -0.027777778 l=0
Found bound state at E=  -0.020407884 E_exact=  -0.020408163 l=0
Found bound state at E=  -0.249999997 E_exact=  -0.250000000 l=1
Found bound state at E=  -0.111111111 E_exact=  -0.111111111 l=1
Found bound state at E=  -0.062500000 E_exact=  -0.062500000 l=1
Found bound state at E=  -0.040000001 E_exact=  -0.040000000 l=1
Found bound state at E=  -0.027777785 E_exact=  -0.027777778 l=1
Found bound state at E=  -0.020407939 E_exact=  -0.020408163 l=1
Found bound state at E=  -0.111111113 E_exact=  -0.111111111 l=2
Found bound state at E=  -0.062500001 E_exact=  -0.062500000 l=2
Found bound state at E=  -0.040000000 E_exact=  -0.040000000 l=2
Found bound state at E=  -0.027777785 E_exact=  -0.027777778 l=2
Found bound state at E=  -0.020408364 E_exact=  -0.020408163 l=2
Found bound state at E=  -0.062500000 E_exact=  -0.062500000 l=3
Found bound state at E=  -0.040000000 E_exact=  -0.040000000 l=3
Found bound state at E=  -0.027777780 E_exact=  -0.027777778 l=3
Found bound state at E=  -0.020408140 E_exact=  -0.020408163 l=3
Found bound state at E=  -0.040000000 E_exact=  -0.040000000 l=4
Found bound state at E=  -0.027777778 E_exact=  -0.027777778 l=4
Found bound state at E=  -0.020408241 E_exact=  -0.020408163 l=4
Found bound state at E=  -0.027777778 E_exact=  -0.027777778 l=5
Found bound state at E=  -0.020408180 E_exact=  -0.020408163 l=5
```

In [12]:
```
def cmpE(x,y):
    if abs(x[1]-y[1])>1e-4:
        return cmp(x[1],y[1])
    else:
        return cmp(x[0],y[0])


Bnd.sort(cmpE)
```

In [13]:
```
Bnd
```

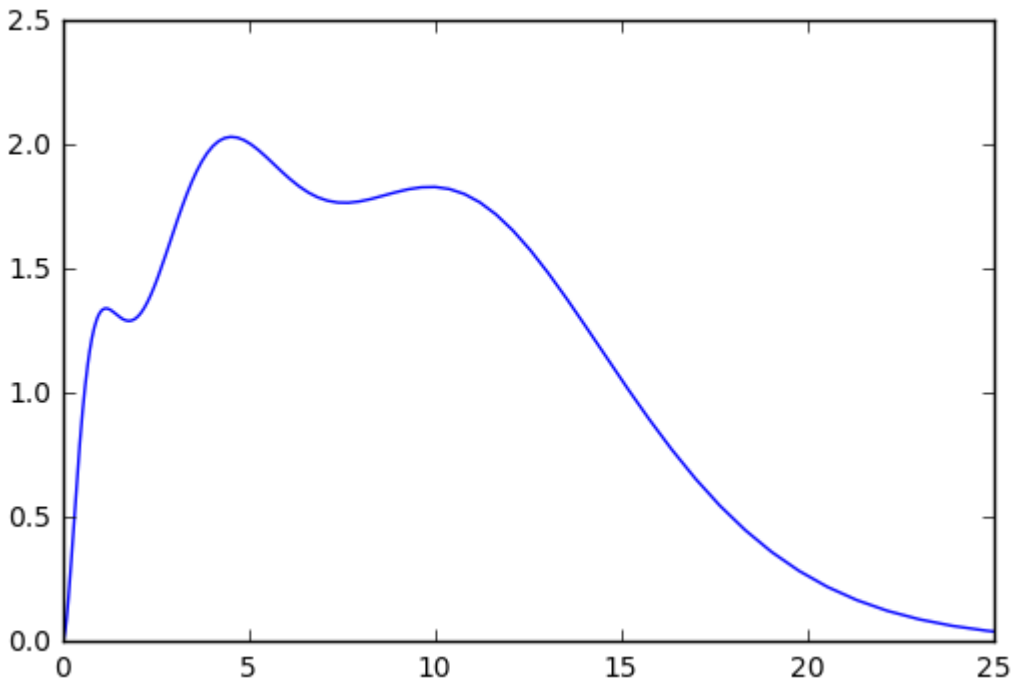
Out[13]:
```
[(0, -1.0000000144346817),
 (0, -0.24999999783074844),
 (1, -0.24999999697029185),
 (0, -0.11111111114527727),
 (1, -0.11111111106834594),
 (2, -0.11111111287698874),
 (0, -0.06250000130575296),
 (1, -0.06250000018929516),
 (2, -0.06250000130055344),
 (3, -0.062499999882876356),
 (0, -0.040000000216110485),
 (1, -0.04000000055775173),
 (2, -0.040000000472501886),
 (3, -0.04000000001991887),
 (4, -0.0399999999299882),
 (0, -0.02777777958583683),
 (1, -0.027777785464469164),
 (2, -0.027777784587779557),
 (3, -0.027777780427529656),
 (4, -0.027777777755184058),
 (5, -0.02777777784154181),
 (0, -0.020407884400230124),
 (1, -0.020407939294276315),
 (2, -0.02040836406577627),
 (3, -0.020408139788464376),
 (4, -0.020408241028337198),
 (5, -0.020408180238043916),
 (0, -0.015566866041478888),
 (1, -0.01557378892995643),
 (2, -0.015585403553970614),
 (3, -0.015598994601132812),
 (4, -0.015609593699604052),
 (5, -0.015617994166722987)]
```

In [14]:
```
Z=28 # like Ni
N=0
rho=zeros(len(R))
for (l,En) in Bnd:
    ur = SolveSchroedinger(En,l,R)
    dN = 2*(2*l+1)
    if N+dN<=Z:
        ferm=1.
    else:
        ferm=(Z-N)/float(dN)
    drho = ur**2 * ferm * dN/(4*pi*R**2)
    rho += drho
    N += dN
    print 'adding state (%2d,%14.9f) with fermi=%4.2f and current N=%5.
1f' % (l,En,ferm,N)
    if N>=Z: break
```

```
adding state ( 0,  -1.000000014) with fermi=1.00 and current N=  2.0
adding state ( 0,  -0.249999998) with fermi=1.00 and current N=  4.0
adding state ( 1,  -0.249999997) with fermi=1.00 and current N= 10.0
adding state ( 0,  -0.111111111) with fermi=1.00 and current N= 12.0
adding state ( 1,  -0.111111111) with fermi=1.00 and current N= 18.0
adding state ( 2,  -0.111111113) with fermi=1.00 and current N= 28.0
```

Resulting charge density for a Ni-like Hydrogen atom

In [15]:
```
from pylab import *
%matplotlib inline

plot(R,rho*(4*pi*R**2),label='charge density')
xlim([0,25])
show()
```



Numerov algorithm

The general purpose integration routine is not the best method for solving the Schroedinger equation, which does not have first derivative terms.

Numerov algorithm is better fit for such equations, and its algorithm is summarized below.

The second order linear differential equation (DE) of the form

$$x''(t) = f(t)x(t) + u(t)$$

is a target of Numerov algorithm.

Due to a special structure of the DE, the fourth order error cancels and leads to sixth order algorithm using second order integration scheme.

If we expand x(t) to some higher power and take into account the time reversal symmetry of the equation, all odd term cancel

$$x(h) = x(0) + hx'(0) + \frac{1}{2}h^2x''(0) + \frac{1}{3!}h^3x^{(3)}(0) + \frac{1}{4!}h^4x^{(4)}(0) + \frac{1}{5!}h^5x^{(5)}(0) + \dots$$
$$x(-h) = x(0) - hx'(0) + \frac{1}{2}h^2x''(0) - \frac{1}{3!}h^3x^{(3)}(0) + \frac{1}{4!}h^4x^{(4)}(0) - \frac{1}{5!}h^5x^{(5)}(0) + \dots$$

hence

$$x(h) + x(-h) = 2x(0) + h^2(f(0)x(0) + u(0)) + \frac{2}{4!}h^4x^{(4)}(0) + O(h^6)$$

If we are happy with $O(h^4)$ algorithm, we can neglect $x^{(4)}$ term and get the following recursion relation

$$x_{i+1} - 2x_i + x_{i-1} = h^2(f_ix_i + u_i).$$

But we know from the differential equation that

$$x^{(4)} = \frac{d^2x''(t)}{dt^2} = \frac{d^2}{dt^2}(f(t)x(t) + u(t))$$

which can be approximated by

$$x^{(4)} \sim \frac{f_{i+1}x_{i+1} + u_{i+1} - 2f_ix_i - 2u_i + f_{i-1}x_{i-1} + u_{i-1}}{h^2}$$

Inserting the fourth order derivative into the above recursive equation (forth equation in his chapter), we get

$$x_{i+1} - 2x_i + x_{i-1} = h^2(f_ix_i + u_i) + \frac{h^2}{12}(f_{i+1}x_{i+1} + u_{i+1} - 2f_ix_i - 2u_i + f_{i-1}x_{i-1} + u_{i-1})$$

If we switch to a new variable $w_i = x_i(1 - \frac{h^2}{12}f_i) - \frac{h^2}{12}u_i$ we are left with the following equation

$$w_{i+1} - 2w_i + w_{i-1} = h^2(f_ix_i + u_i) + O(h^6)$$

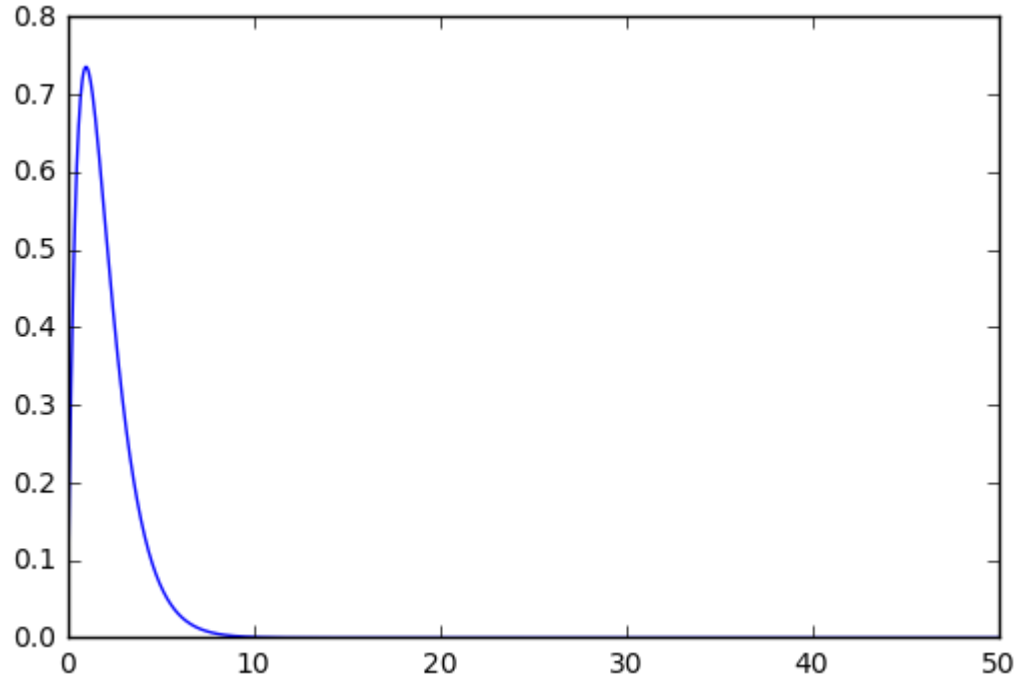
The variable x needs to be recomputed at each step with $x_i = (w_i + \frac{h^2}{12}u_i)/(1 - \frac{h^2}{12}f_i)$.

In [16]:
```
def Numerov(f, x0, dx, dh):
    """Given precomputed function f(x), solves for x(t), which satisfies:
          x''(t) = f(t) x(t)
    """
    x = zeros(len(f))
    x[0] = x0
    x[1] = x0+dh*dx

    h2 = dh**2
    h12 = h2/12.

    w0=x0*(1-h12*f[0])
    w1=x[1]*(1-h12*f[1])
    xi = x[1]
    fi = f[1]
    for i in range(2,len(f)):
        w2 = 2*w1-w0+h2*fi*xi  # here fi=f1
        fi = f[i]  # fi=f2
        xi = w2/(1-h12*fi)
        x[i]=xi
        w0 = w1
        w1 = w2
    return x
```

In [17]:
```
def fSchrod(En, l, R):
    return l*(l+1.)/R**2-2./R-En
```

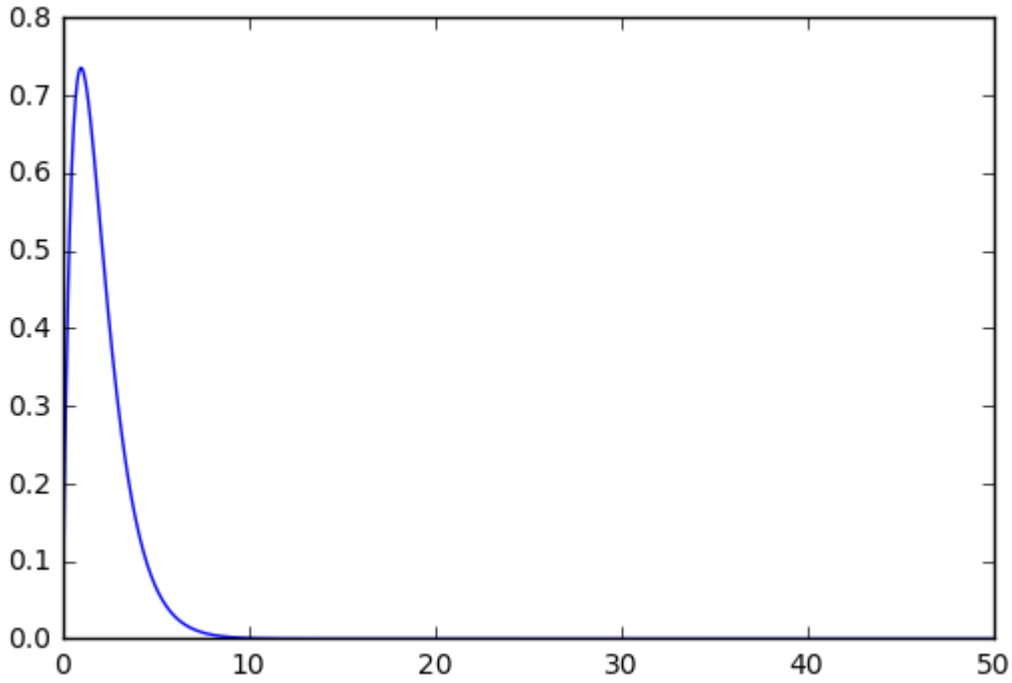
In [18]:
```
Rl = linspace(1e-7,50,1000)
l=0
En=-1.
f = fSchrod(En,l,Rl[::-1])
ur = Numerov(f,0.0,1e-7,Rl[1]-Rl[0])[::-1]
norm = integrate.simps(ur**2,x=Rl)
ur *= 1/sqrt(abs(norm))
```

In [19]:
```
from pylab import *
%matplotlib inline

plot(Rl,ur)
show()
```



In [22]:
```
import weave
def Numerovc(f, x0_, dx, dh_):
    code_Numerov="""
    double h2 = dh*dh;
    double h12 = h2/12.;

    double w0 = x(0)*(1-h12*f(0));
    double w1 = x(1)*(1-h12*f(1));
    double xi = x(1);
    double fi = f(1);
    for (int i=2; i<f.size(); i++){
        double w2 = 2*w1-w0+h2*fi*xi;  // here fi=f1
        fi = f(i);                     // fi=f2
        xi = w2/(1-h12*fi);
        x(i)=xi;
        w0 = w1;
        w1 = w2;
    }
    """
    x = zeros(len(f))
    dh=float(dh_)
    x[0]=x0_
    x[1]=x0_+dh*dx
    weave.inline(code_Numerov, ['f','dh','x'], type_converters=weave.co
nverters.blitz, compiler = 'gcc')
    return x
```

In [23]:
```
Rl = linspace(1e-7,50,1000)
l=0
En=-1.
f = fSchrod(En,l,Rl[::-1])
ur = Numerovc(f,0.0,1e-7,Rl[1]-Rl[0])[::-1]
norm = integrate.simps(ur**2,x=Rl)
ur *= 1/sqrt(abs(norm))

plot(Rl,ur)
show()
```

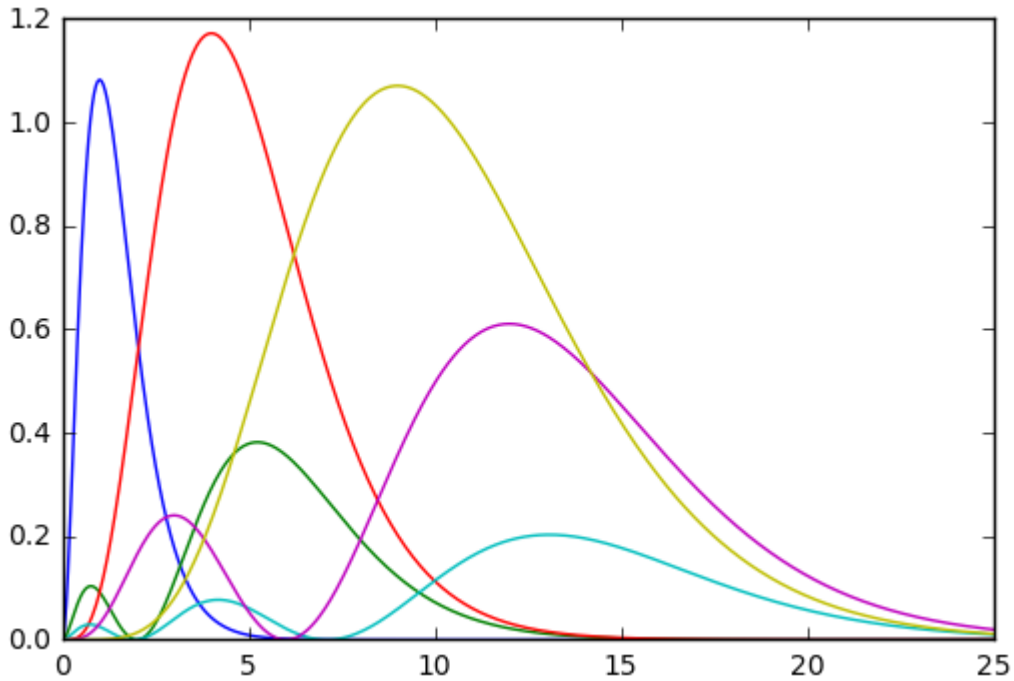


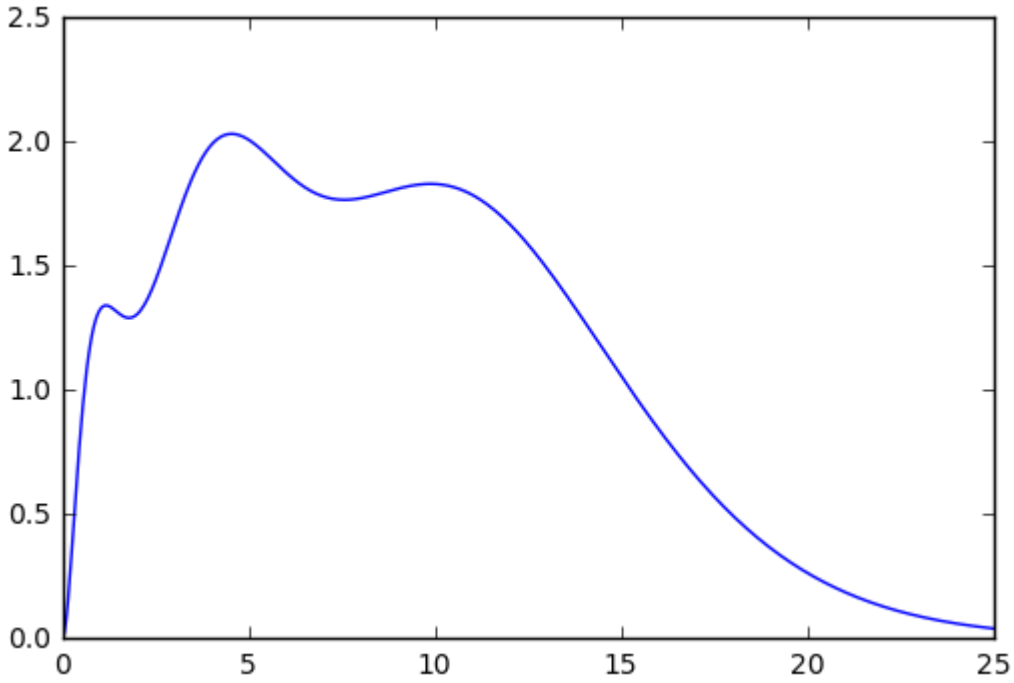
Put it all together

In [26]:
```
import weave
def Numerovc(f, x0_, dx, dh_):
    code_Numerov="""
    double h2 = dh*dh;
    double h12 = h2/12.;

    double w0 = x(0)*(1-h12*f(0));
    double w1 = x(1)*(1-h12*f(1));
    double xi = x(1);
    double fi = f(1);
    for (int i=2; i<f.size(); i++){
        double w2 = 2*w1-w0+h2*fi*xi;  // here fi=f1
        fi = f(i);                     // fi=f2
        xi = w2/(1-h12*fi);
        x(i)=xi;
        w0 = w1;
        w1 = w2;
    }
    """
    x = zeros(len(f))
    dh=float(dh_)
    x[0]=x0_
    x[1]=x0_+dh*dx
    weave.inline(code_Numerov, ['f','dh','x'], type_converters=weave.co
nverters.blitz, compiler = 'gcc')
    return x

def fSchrod(En, l, R):
    return l*(l+1.)/R**2-2./R-En

def ComputeSchrod(En,R,l):
    "Computes Schrod Eq."
    f = fSchrod(En,l,R[::-1])
    ur = Numerovc(f,0.0,-1e-7,-R[1]+R[0])[::-1]
    norm = integrate.simps(ur**2,x=R)
    return ur*1/sqrt(abs(norm))

def Shoot(En,R,l):
    ur = ComputeSchrod(En,R,l)
    ur = ur/R**l
    f0 = ur[0]
    f1 = ur[1]
    f_at_0 = f0 + (f1-f0)*(0.0-R[0])/(R[1]-R[0])
    return f_at_0

def FindBoundStates(R,l,nmax,Esearch):
    n=0
    Ebnd=[]
    u0 = Shoot(Esearch[0],R,l)
    for i in range(1,len(Esearch)):
        u1 = Shoot(Esearch[i],R,l)
        if u0*u1<0:
            Ebound = optimize.brentq(Shoot,Esearch[i-1],Esearch[i],xtol
=1e-16,args=(R,l))
            Ebnd.append((l,Ebound))
            if len(Ebnd)>nmax: break
```

In [27]:
```
def cmpE(x,y):
    if abs(x[1]-y[1])>1e-4:
        return cmp(x[1],y[1])
    else:
        return cmp(x[0],y[0])
Esearch = -1.2/arange(1,20,0.2)**2
R = linspace(1e-8,100,2000)
nmax=5
Bnd=[]
for l in range(nmax-1):
    Bnd += FindBoundStates(R,l,nmax-l,Esearch)

Bnd.sort(cmpE)

Z=28  # Like Ni ion

N=0
rho=zeros(len(R))
for (l,En) in Bnd:
    #ur = SolveSchroedinger(En,l,R)
    ur = ComputeSchrod(En,R,l)
    dN = 2*(2*l+1)
    if N+dN<=Z:
        ferm=1.
    else:
        ferm=(Z-N)/float(dN)
    drho = ur**2 * ferm * dN/(4*pi*R**2)
    rho += drho
    N += dN
    print 'adding state', (l,En), 'with fermi=', ferm
    plot(R, drho*(4*pi*R**2))
    if N>=Z: break
xlim([0,25])
show()

plot(R,rho*(4*pi*R**2),label='charge density')
xlim([0,25])
show()
```

```
Found bound state at E=  -0.999922109 E_exact=  -1.000000000 l=0
Found bound state at E=  -0.249990190 E_exact=  -0.250000000 l=0
Found bound state at E=  -0.111108201 E_exact=  -0.111111111 l=0
Found bound state at E=  -0.062498772 E_exact=  -0.062500000 l=0
Found bound state at E=  -0.039999314 E_exact=  -0.040000000 l=0
Found bound state at E=  -0.250000016 E_exact=  -0.250000000 l=1
Found bound state at E=  -0.111111117 E_exact=  -0.111111111 l=1
Found bound state at E=  -0.062500003 E_exact=  -0.062500000 l=1
Found bound state at E=  -0.039999959 E_exact=  -0.040000000 l=1
Found bound state at E=  -0.111111111 E_exact=  -0.111111111 l=2
Found bound state at E=  -0.062500000 E_exact=  -0.062500000 l=2
Found bound state at E=  -0.039999977 E_exact=  -0.040000000 l=2
Found bound state at E=  -0.062500000 E_exact=  -0.062500000 l=3
Found bound state at E=  -0.039999992 E_exact=  -0.040000000 l=3
adding state (0, -0.9999221089559618) with fermi= 1.0
adding state (0, -0.24999019020652996) with fermi= 1.0
adding state (1, -0.25000001561170354) with fermi= 1.0
adding state (0, -0.11110820082299919) with fermi= 1.0
adding state (1, -0.11111111678092289) with fermi= 1.0
adding state (2, -0.11111111114690239) with fermi= 1.0
```





In [ ]:

In [ ]: