Hydrogen Atom

- **Author:** Ariel Quelal
- **Date:** 12/09/2025
- **Time spent on this assignment:** 50h

In [1]:
```
import numpy as np
from numpy import *
import matplotlib.pyplot as plt
import scipy
import scipy.sparse
import scipy.sparse.linalg
import plotly.graph_objects as go
from scipy.constants import physical_constants
import scipy.special as sp
from mpl_toolkits.mplot3d import Axes3D
import math
```

Throughout this exercise, we are going to work in atomic units where $\hbar = m = e = 1/(4\pi\epsilon_0) = 1$. This gives us the Bohr radius which is also equal to 1 in these units (and 0.529×10^{-10} m in SI units). All distances in this unit will be measured in Bohr radii.

Exercise 1. Plotting the Hydrogen Atom Orbitals

In this exercise, we are going to plot the orbitals of the Hydrogen atom starting from the relevant solutions that you found analytically. In future exercises, we will see how to get these results from scratch without using the analytic solutions.

Recall that the Hydrogen atom wave-function is

$$\Psi_{nlm}(r, \theta, \phi) = R_{nl}(r) Y_{lm}(\theta, \phi)$$

a. Radial Function

Let us start by getting the radial wave-function of the Hydrogen atom working. Recall that the radial wave-function is

$$R_{nl}(r) = \frac{u_{nl}(r)}{r} = N \exp(-\rho/2) \rho^l L_{2l+1}^{n-l-1}(\rho)$$

where

$$\rho = \frac{2r}{n a_0}$$

and

$$N = \sqrt{\left(\frac{2}{na_0}\right)^3 \frac{(n-l-1)!}{2n(n+l)!}}$$

and L is a special function which can be generated by doing `L=sp.genlaguerre(n-l-1,2*l+1)`. Note that this just returns the python function. Later you will have to call `L(rho)`.

Write a function `radial_function(n, l, r)` which takes n, l, and r and returns $R_{nl}(r)$. Recall we are in units where $a_0 = 1$.

Let's start by evaluating this on a one-dimensional grid of positions `r=np.linspace(0.001,45,1000)`.

Plot on the same plot, $u_{nl}(r)$ (notice this is different from R_{nl} by a factor of r) for

- $(n, \ell) = (1, 0)$
- $(n, \ell) = (2, 0)$
- $(n, \ell) = (3, 0)$

How do the number of nodes (times when it hits zero) correspond to the value of n?

Now on a separate plot make

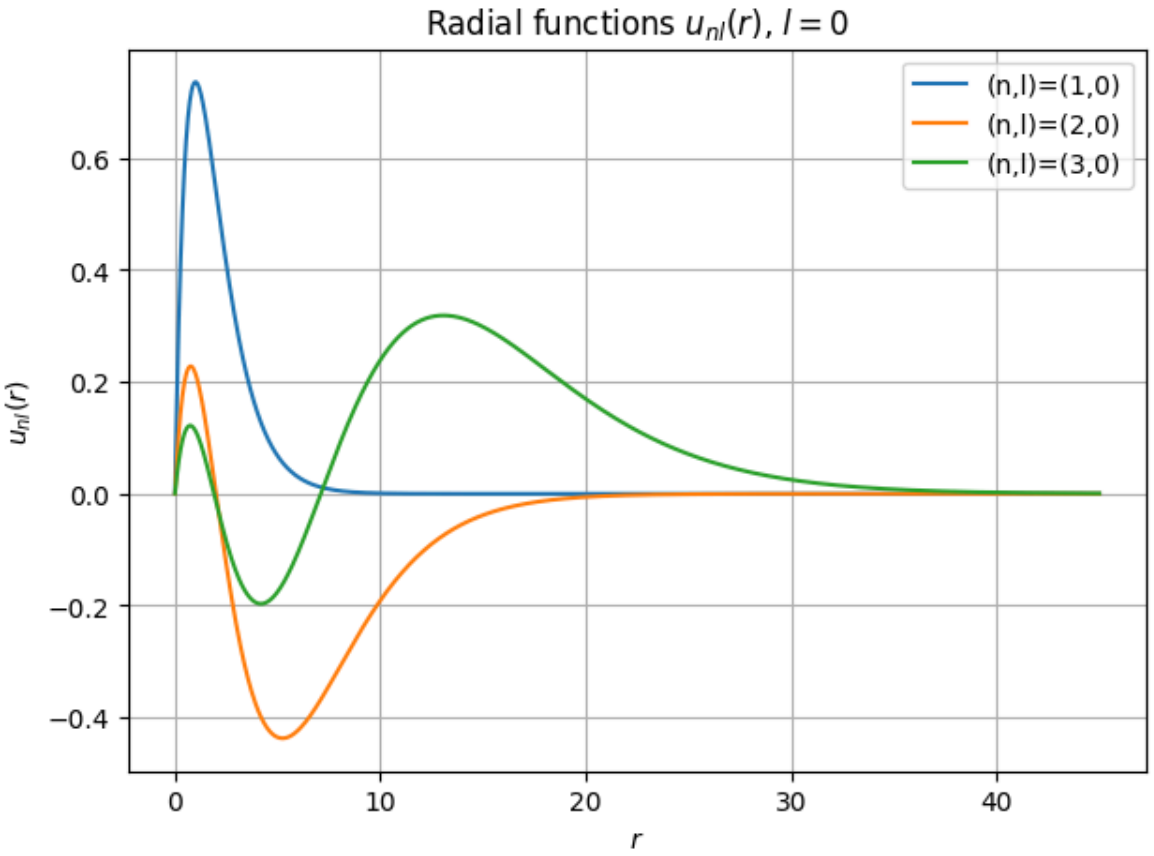
- $(n, \ell) = (3, 2)$
- $(n, \ell) = (3, 1)$
- $(n, \ell) = (3, 0)$

Notice how the larger values of l are pushed further away from the origin. This is coming from the additional force due to the fake potential.

? Answer (start)

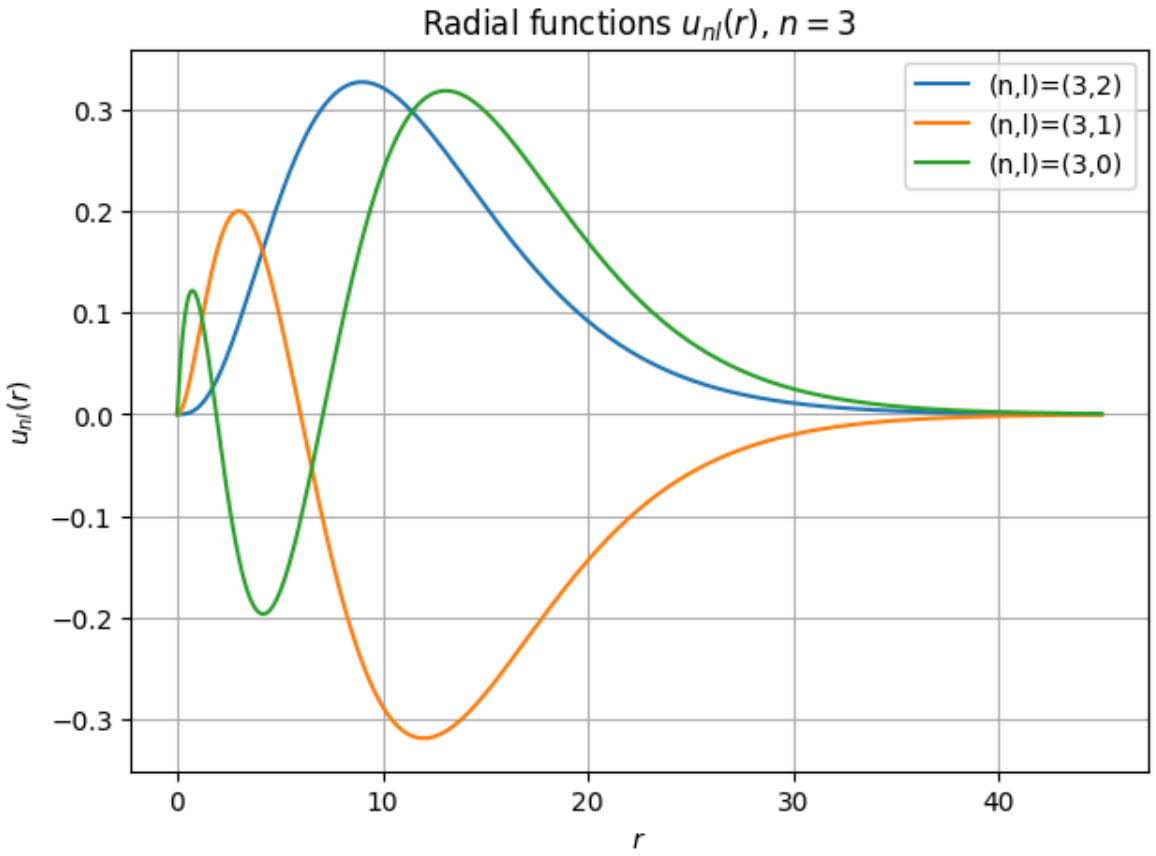
In [2]:
```
#Define the radial function R_{n l}(r)

def radial_function(n, l, r):
    # rho = 2r/(n a0), a0=1
    rho = 2*r/n
    # Associated Laguerre polynomial L_{n-l-1}^{(2l+1)}(rho)
    L = sp.genlaguerre(n - l - 1, 2*l + 1)
    # Normalization constant
    N = np.sqrt((2/n)**3 * math.factorial(n - l - 1) /
                (2*n * math.factorial(n + l)))

    R = N * np.exp(-rho/2) * rho**l * L(rho)
    return R

#Set up the grid for variable r

r = np.linspace(0.001, 45, 1000)

#Plot u_{n l}(r) = r * R_{n l}(r)
```

```
plt.figure(figsize=(7,5))

for n in [1, 2, 3]:
    l = 0
    R = radial_function(n, l, r)
    u = r * R
    plt.plot(r, u, label=f"(n,l)=({n},{l})")

plt.xlabel("$r$")
plt.ylabel("$u_{nl}(r)$")
plt.title("Radial functions $u_{nl}(r)$, $l=0$")
plt.legend()
plt.grid(True)
plt.show()
```



The number of times $u_{n0}(r)$ hits zero is equal to $\text{number of nodes} = n - 1$. (Recall that $r \neq 0$)

In [3]:
```
plt.figure(figsize=(7,5))

for l in [2, 1, 0]:
    n = 3
    R = radial_function(n, l, r)
    u = r * R
    plt.plot(r, u, label=f"(n,l)=({n},{l})")

plt.xlabel("$r$")
plt.ylabel("$u_{nl}(r)$")
plt.title("Radial functions $u_{nl}(r)$, $n=3$")
plt.legend()
plt.grid(True)
plt.show()
```



It seems to be the case that the number of nodes are given by:

$$\text{number of nodes} = n - l - 1$$

✓ Answer (end)

It should also be that the different $u_{nl}(r)$ at fixed l are orthogonal to each other. Check this by verifying that `u31.conjugate() @ u21.T=0` where you get `u` by multiplying your radial function by r.

? Answer (start)

In [4]:
```
#Compute u_nl(r)
u31 = r * radial_function(3, 1, r)
u21 = r * radial_function(2, 1, r)

#Inner product
dr=r[1]-r[0]
inner = u31.conjugate() @ u21.T * dr #dr is used because we are approximating an
print("<u31|u21> =", inner)
```

```
<u31|u21> = 5.0578572732501776e-11
```

✓ Answer (end)

b. Hydrogen Orbitals

We can now combine the radial wave-function with the spherical Harmonics to write down the Hydrogen orbitals

$$\Psi_{nlm}(x,y,z) = R_{nl}(x,y,z)Y_{lm}(\theta,\phi)$$

We are going to want to evaluate these orbitals on the entire space. Therefore, we will start by putting together a three-dimensional grid of the (x,y,z) positions.

```
def HydrogenGrid(g=10,numGridPoints=50):
    p=np.linspace(-g,g,numGridPoints)
    h=p[1]-p[0]
    x,y,z=np.meshgrid(p,p,p)
    return x,y,z,h
```

which you can then get the grid from by doing `x,y,z,h=HydrogenGrid(g)`

You'll adjust the grid length `g` depending on which orbital you're trying to plot. This will return three three-dimensional arrays x,y, z (as well as the grid spacing h).

The three-dimensional array x simply has the coordinate of x at every point in (the discretized) 3D space and the three-dimensional array y has the coordinate of y at every point in (the discretized) 3D space - e.g. the analogous 2D version of this are $x =$

```
[[-8. -4.  0.  4.  8.]
 [-8. -4.  0.  4.  8.]
 [-8. -4.  0.  4.  8.]
 [-8. -4.  0.  4.  8.]
 [-8. -4.  0.  4.  8.]]
```

and $y =$

```
[[-8. -8. -8. -8. -8.]
 [-4. -4. -4. -4. -4.]
 [ 0.  0.  0.  0.  0.]
 [ 4.  4.  4.  4.  4.]
 [ 8.  8.  8.  8.  8.]]
```

Python generates grids like this using `np.meshgrid` (see the `HydrogenGrid` function)

By using these three grids, it makes it easy to produce three-dimensional grids of the values of r or θ or ϕ.

Go ahead and write a function `Cartesian2Spherical(x,y,z)` which returns three-dimensional grids of r, θ, and ϕ at every point in space. Recall that

$$\theta = \tan^{-1}(y/x) + \pi$$

(use `np.arctan2(y,x)`)

and

$$\phi = \cos^{-1}(z/r)$$

You should now have access to r, θ, ϕ on your grid from which you can call both your radial function and spherical harmonics.

? Answer (start)

In [5]:
```
#Hydrogen 3D grid function

def HydrogenGrid(g=10,numGridPoints=50):
    p=np.linspace(-g,g,numGridPoints)
    h=p[1]-p[0]
    x,y,z=np.meshgrid(p,p,p)
    return x,y,z,h

#Cartesian to spherica coordinate transformation

import numpy as np

def Cartesian2Spherical(x, y, z):
    # Radius
    r = np.sqrt(x**2 + y**2 + z**2)
    # Azimuthal angle theta in [-pi, pi]
    theta = np.arctan2(y, x)
    # Polar angle phi in [0, pi]; guard against division by zero
    phi = np.zeros_like(r)
    # Avoid warnings at r = 0
    with np.errstate(invalid='ignore', divide='ignore'):
        phi = np.arccos(np.clip(z / r, -1.0, 1.0)) #Clip mantains the values wit
    # Define phi arbitrarily at r = 0 (it doesn't matter physically there)
    phi[r == 0] = 0.0

    return r, theta, phi
```

✓ Answer (end)

Now write a function `HydrogenAtom(n, l, m,x,y,z)` which computes the hydrogen orbital on this grid and returns `psi`.

The spherical Harmonics is

```
sp.sph_harm_y(l,m,phi,theta)
```

where the m and l are indexing the eigenstates of the angular momentum operator. (Notice that this is backwards from how you typical think about the quantum numbers for the hydrogen atom).

To plot the wave-function use the following

```
def PlotMe(psi,x,y,z):
    isosurface = go.Isosurface(
        x=x.flatten(),
        y=y.flatten(),
        z=z.flatten(),
        value=(np.real(psi)**2).flatten(),  # Use the absolute
value for intensity
        surface_count=20, # Number of isosurfaces to display
        opacity=0.5,  # Set the opacity of the isosurface
        surface_fill=1.0,
        caps=dict(x_show=False, y_show=False,z_show=False)
    )
```

```
    fig = go.Figure(data=[isosurface])
    fig.show()
```

Plot $(n, \ell, m) = (1, 0, 0); (2, 1, 0); (3, 2, 2); (3, 2, 0)$ where the grid-length I used to get a good figure was

```
g=dict()
g[(1,0,0)]=4
g[(2,1,0)]=7.7
g[(3,2,2)]=18
g[(3,2,0)]=20
```

? Answer (start)

In [6]:
```
#Define the HydrogenAtom function

def HydrogenAtom(n, l, m, x, y, z):
    #Cartesian to spherical coordinates
    r, theta, phi = Cartesian2Spherical(x, y, z)
    #Radial part R_{nl}(r)
    R = radial_function(n, l, r)
    #Angular part Y_{lm}(theta, phi) -> Spherical Harmonics
    Y = sp.sph_harm_y(l, m, phi, theta)
    #Wave function value function Psi = R(r) * Y_lm(theta, phi)
    psi = R * Y
    return psi
```

In [7]:
```
#Plotting function
def PlotMe(psi,x,y,z):
    isosurface = go.Isosurface(
        x=x.flatten(),
        y=y.flatten(),
        z=z.flatten(),
        value=(np.real(psi)**2).flatten(),  # Use the absolute value for intensi
        surface_count=20, # Number of isosurfaces to display
        opacity=0.5,  # Set the opacity of the isosurface
        surface_fill=1.0,
        caps=dict(x_show=False, y_show=False,z_show=False)
    )

    fig = go.Figure(data=[isosurface])
    fig.show()
```

In [8]:
```
#Setting uo the grid sizes
g=dict()
g[(1,0,0)]=4
g[(2,1,0)]=7.7
g[(3,2,2)]=18
g[(3,2,0)]=20
```

In [9]:
```
#Plotting the Hydrogen atom
for (n, l, m), gval in g.items():
    print(f"Orbital (n,l,m)=({n},{l},{m}) with g={gval}")
    x, y, z, h = HydrogenGrid(g=gval, numGridPoints=50)
    psi = HydrogenAtom(n, l, m, x, y, z)
    PlotMe(psi, x, y, z)
```

```
Orbital (n,l,m)=(1,0,0) with g=4
Orbital (n,l,m)=(2,1,0) with g=7.7
Orbital (n,l,m)=(3,2,2) with g=18
Orbital (n,l,m)=(3,2,0) with g=20
```

LOOK AT THE END OF THE DOCUMENT TO FIND THE IMAGES OF THE 3D PLOTS!

✓ Answer (end)

c. Getting the Energy of the Hydrogen Atom

Now we are going to compute the energy of the Hydrogen atom. Recall that the Hamiltonian for the Hydrogen atom is

$$H = -\frac{1}{2}\nabla^2 + \frac{1}{r}$$

where $-\nabla^2 = \partial^2/\partial x^2 + \partial^2/\partial y^2 + \partial^2/\partial z^2$ and r is the distance between the nucleus and the electron.

Let's figure out how to compute each of these terms.

This second term can be computed by taking the integral

$$\int \Psi(x,y,z)\left(-\frac{1}{r}\right)\Psi^*(x,y,z)dxdxdz$$

If we evaluate our wave-function `psi` on the grid (which is how we got our orbitals that we visualize above), then we simply multiply these three terms, sum them up (`np.sum`) and then multiply by our grid spacing cubed.

For the $(n,l,m) = (1,0,0)$ you should get a potential Energy of -0.9897509703021152 using the following grid `x,y,z,delta_x=HydrogenGrid(20,200)`

Implement this function and make sure you get the correct energy.

? Answer (start)

In [10]:
```
#Set up the grid
x,y,z,delta_x=HydrogenGrid(20,200)

#Potential energy function
def potential_energy(psi, x, y, z, dx):
    #Radius
    r = np.sqrt(x**2 + y**2 + z**2)
    #Define potential V = -1/r, but avoiding division by zero at r=0
    V = np.zeros_like(r)
    nonzero = (r != 0)
    V[nonzero] = -1.0 / r[nonzero]

    # Probability density |psi|^2
    prob = np.abs(psi)**2

    # Discrete approximation to the integral: sum(psi* V psi) dx^3
    dV = dx**3
    V_expect = np.sum(prob * V) * dV

    return V_expect
```

```
#Compute potential energy for state (n,m,l) = (1,0,0)
psi = HydrogenAtom(1, 0, 0, x, y, z)
V = potential_energy(psi, x, y, z, delta_x)

print("V = ", V)
```

```
V =  -0.9897509703021151
```

✓ Answer (end)

Now we need to get the next term

$$-\frac{1}{2}\int \Psi^*(x,y,z)\frac{\partial^2}{\partial x^2}\Psi(x,y,z)dxdxdz$$

We can compute this by finite differences. The standard formula for the second derivative is

$$\frac{\partial^2}{\partial x^2} = \frac{f(x+dx)+f(x-dx)-2f(x)}{dx^2}$$

Write a function `Laplacian(n,l,m)` which calls `HydrogenWaveFunction` three times (per direction) returning the laplacian; for example for the x-direction it calls `x+dx`, `x-dx`, and `x`. For `dx` use 1e-3.

Compute the kinetic energy of the $(n,l,m) = (1,0,0)$ state. You should get 0.4897835961971023 total with each of the three kinetic terms being the same (because of the symmetry of this state)

Now sum up the kinetic and potential energies and multiply by 27.2 to get it into eV. Verify that you get the right answer for the (1,0,0) term. Also compute the energies for the (2,0,0), (2,1,0), (2,1,1) and (3,2,1) terms. Check that each of them are close to the known value.

? Answer (start)

In [11]:
```
#Define Laplacian using finite differences in each direction, dx_fd is the step
def Laplacian(n, l, m, x, y, z, dx_fd=1e-3):
    #Wavefunction
    psi0 = HydrogenAtom(n, l, m, x, y, z)

    #Second derivative in x
    psi_x_plus  = HydrogenAtom(n, l, m, x + dx_fd, y, z)
    psi_x_minus = HydrogenAtom(n, l, m, x - dx_fd, y, z)
    d2psi_dx2 = (psi_x_plus + psi_x_minus - 2*psi0) / dx_fd**2

    #Second derivative in y
    psi_y_plus  = HydrogenAtom(n, l, m, x, y + dx_fd, z)
    psi_y_minus = HydrogenAtom(n, l, m, x, y - dx_fd, z)
    d2psi_dy2 = (psi_y_plus + psi_y_minus - 2*psi0) / dx_fd**2

    #Second derivative in z
    psi_z_plus  = HydrogenAtom(n, l, m, x, y, z + dx_fd)
    psi_z_minus = HydrogenAtom(n, l, m, x, y, z - dx_fd)
    d2psi_dz2 = (psi_z_plus + psi_z_minus - 2*psi0) / dx_fd**2
```

```
    lapPsi = d2psi_dx2 + d2psi_dy2 + d2psi_dz2
    return lapPsi, d2psi_dx2, d2psi_dy2, d2psi_dz2 #Return laplacian, and secona

#Kinetic energy function
def kinetic_energy_components(n, l, m, x, y, z, delta_x, dx_fd=1e-3):

    #Wavefunction
    psi0 = HydrogenAtom(n, l, m, x, y, z)

    #Laplacian
    lapPsi, d2psi_dx2, d2psi_dy2, d2psi_dz2 = Laplacian(n, l, m, x, y, z, dx_fd)

    #Measure
    dV = delta_x**3

    #Kinetic energy x component
    Tx = -0.5 * np.sum(np.conjugate(psi0) * d2psi_dx2) * dV
    Tx=real(Tx)

    #Kinetic energy y component
    Ty = -0.5 * np.sum(np.conjugate(psi0) * d2psi_dy2) * dV
    Ty=real(Ty)

    #Kinetic energy y component
    Tz = -0.5 * np.sum(np.conjugate(psi0) * d2psi_dz2) * dV
    Tz=real(Tz)
    #Total kinetic energy
    T_total = -0.5 * np.sum(np.conjugate(psi0) * lapPsi) * dV
    T_total=real(T_total)
      # It could also have been computed with T_total = Tx + Ty + Tz, but let's

    return Tx, Ty, Tz, T_total
```

In [12]:

```
#Compute the required values for the desired states
def total_energy_state(n, l, m):
    psi = HydrogenAtom(n, l, m, x, y, z)
    V = potential_energy(psi, x, y, z, delta_x)
    Tx, Ty, Tz, T = kinetic_energy_components(n, l, m, x, y, z, delta_x)
    E = T + V
    E_eV = E * 27.2         # convert to eV
    return V, (Tx, Ty, Tz, T), E, E_eV

states = [
    (1, 0, 0),
    (2, 0, 0),
    (2, 1, 0),
    (2, 1, 1),
    (3, 2, 1),
]

for (n, l, m) in states:
    V, (Tx, Ty, Tz, T), E, E_eV = total_energy_state(n, l, m)
    print(f"(n,l,m)=({n},{l},{m})")
    print(f"  Potential energy V    ≈ {V}")
    print(f"  Kinetic components Tx,Ty,Tz ≈ {Tx}, {Ty}, {Tz}")
    print(f"  Total kinetic T       ≈ {T}")
    print(f"  Total energy E (a.u.) ≈ {E}")
```

```
    print(f"  Total energy (eV)     ≈ {E_eV}")
    print()
```

```
(n,l,m)=(1,0,0)
  Potential energy V    ≈ -0.9897509703021151
  Kinetic components Tx,Ty,Tz ≈ 0.16326119873464473, 0.1632611987346447, 0.163261
1987338868
  Total kinetic T       ≈ 0.489783596203177
  Total energy E (a.u.) ≈ -0.4999673740989381
  Total energy (eV)     ≈ -13.599112575491116

(n,l,m)=(2,0,0)
  Potential energy V    ≈ -0.24871759671413457
  Kinetic components Tx,Ty,Tz ≈ 0.04123981986648398, 0.041239819866484125, 0.0412
3981986601198
  Total kinetic T       ≈ 0.12371945959898005
  Total energy E (a.u.) ≈ -0.12499813711515452
  Total energy (eV)     ≈ -3.3999493295322027

(n,l,m)=(2,1,0)
  Potential energy V    ≈ -0.25000022052560705
  Kinetic components Tx,Ty,Tz ≈ 0.024999463757696294, 0.024999463757696294, 0.075
00160668021426
  Total kinetic T       ≈ 0.12500053419560667
  Total energy E (a.u.) ≈ -0.12499968633000039
  Total energy (eV)     ≈ -3.3999914681760104

(n,l,m)=(2,1,1)
  Potential energy V    ≈ -0.25000022052560733
  Kinetic components Tx,Ty,Tz ≈ 0.05000053522000014, 0.050000535220014033, 0.0249
99463759045097
  Total kinetic T       ≈ 0.1250005341990593
  Total energy E (a.u.) ≈ -0.12499968632654804
  Total energy (eV)     ≈ -3.3999914680821064

(n,l,m)=(3,2,1)
  Potential energy V    ≈ -0.11097107297458254
  Kinetic components Tx,Ty,Tz ≈ 0.01588523514127186, 0.015885235141411824, 0.0238
3968257004509
  Total kinetic T       ≈ 0.05561015285272872
  Total energy E (a.u.) ≈ -0.05536092012185383
  Total energy (eV)     ≈ -1.505817027314424
```

✓ Answer (end)

Exercise 2. Solving the Hydrogen Atom Radial Equation Numerically

Recall that the Hydrogen atom Hamiltonian is

$$H = -\frac{1}{2}\nabla^2 - \frac{1}{r}$$

We solve the hydrogen atom using separation of variables eventually finding that

$$\Psi(R, \theta, \phi) = R_{nl}(r)Y_{lm}(\theta, \phi)$$

where $R_{nl}(r) = u(r)/r$ and $u(r)$ obeys the raidial eigenvalue equation

$$\left[-\frac{1}{2}\left(\frac{d^2}{dr^2} - \frac{l(l+1)}{r^2}\right) - \frac{1}{r}\right] u(r) = Eu(r)$$

which we can write as

$$H_{eff}u(r) = Eu(r)$$

This problem looks a lot like the particle in the box. We can think of r here just like the x coordinate. Then the Hamiltonian has three terms:

- the standard kinetic energy term
- a potential term coming from the angular momentum
- a potential term coming from the coulomb potential.

To solve this analytically required a significant amount of work. We will see that solving it numerically is relatively straightforward. For each of the terms in H_{eff} we need to write out a $N \times N$ Hamiltonian matrix (where N is the number of points in our grid which discretizes space).

a. The Kinetic Term

We will start by working with a discretized grid which is $N = 200$ points defined by

```
N=200
r = np.linspace(40, 0.0, N, endpoint=False)
```

Later to actually get accurate answers, we are going to want to increase N which will actually require us to make some changes (moving to sparse matrices) but let's start with the smaller dense matrices.

We are building $N \times N$ matrices where each row (respectively column) corresponds to a values of r on the grid - i.e. the "third" row corresponds to r[2] and the "fourth" column corresponds to the value of r[3].

Let's start with the term

$$\frac{d^2}{dr^2} = \frac{2f(r) - f(r+dr) - f(r-dr)}{dr^2}$$

As a matrix this means:

- the elements that connect the r'th column to the r'th row should get a $2/dr^2$. This is just the diagonal of the matrix
- the elements that connect the (r+1)'st column to the r'th row should get a $-1/dr^2$. This is just one row above the diagonal
- the elements that connect the (r-1)'st column to the r'th row should get a $-1/dr^2$. This is just one row below the diagonal.

Write a function `Laplacian(r)` which takes the grid `r` and returns a matrix corresponding to the second derivative. You will find the `np.diag` function useful (even the off-diagonal term can be built using it - look at the documentation!). You should essentially be able to build this matrix in three such function calls.

Working in units where $\hbar = 1$ and $m = 1$, go ahead and compute the full kinetic term. To diagonalize a matrix, you can call `e,v=np.linalg.eigh(-0.5*L)`. Verify that the lowest four energies are [0.00305358 0.01221356 0.02747771 0.04884231].

? Answer (start)

In [13]:
```
#Note: The instruction has a typo defining the second derivative d2/dr2, it is g

#Set up grid for r
N = 200
r = np.linspace(40, 0.0, N, endpoint=False)

#Define Laplacian operator, actually just a second derivative operator
def Laplacian(r):
    N = len(r)
    dr = r[1] - r[0]          # grid spacing (negative here, but dr**2 > 0)
    coeff = 1.0 / (dr**2)

    # main diagonal: 2/dr^2
    main_diag = -2.0 * coeff * np.ones(N)

    # off-diagonals: -1/dr^2
    off_diag = 1.0 * coeff * np.ones(N - 1)

    # build the tridiagonal matrix
    L = (
        np.diag(main_diag, k=0) +
        np.diag(off_diag,  k=1) +
        np.diag(off_diag,  k=-1)
    )
    return L

#Computing the value of the kinetic energy term

L = Laplacian(r)
e, v = np.linalg.eigh(-0.5 * L)

print(e[:4])
```

```
[0.00305358 0.01221356 0.02747771 0.04884231]
```

✓ Answer (end)

b. The potential terms

The two potential terms are a function of r and ther matrices for these two terms therefore only connect the r'th row to the r'th column. This means both additional terms only exist on the diagonal. Build a function `V(r,l)` that takes the `r` grid and returns the matrix corresponding to the potential

$$V(r,l) = \frac{l(l+1)}{2r^2} - \frac{1}{r}$$

Again use `np.diag`

❓ Answer (start)

In [14]:
```
#Define the potential operator
def V(r, l):
    # Compute the potential values on the grid: centrifugal + Coulomb
    V_values = l*(l+1)/(2 * r**2) - 1.0/r

    # Build diagonal matrix
    Veff = np.diag(V_values)
    return Veff
```

✅ Answer (end)

c. Putting it together

Now we simply want to add the full Hamiltonian matrix together and diagonalize it using `e,v=np.linalg.eigh(H)`

Both print and plot the lowest 6 energies - `plt.plot(e[0:6]*27.2)`. Also on your plot make some straight lines - `plt.axhline(the_energy)` where the energies for the n'th eignenstate should be - i.e. $-13.6/n^2$. How close are you getting?

Also plot the lowest three eigenstates and see if the radial functions match the ones from above. To normalize the plot correctly you have to divide your eigenvectors by $\sqrt{h}$.

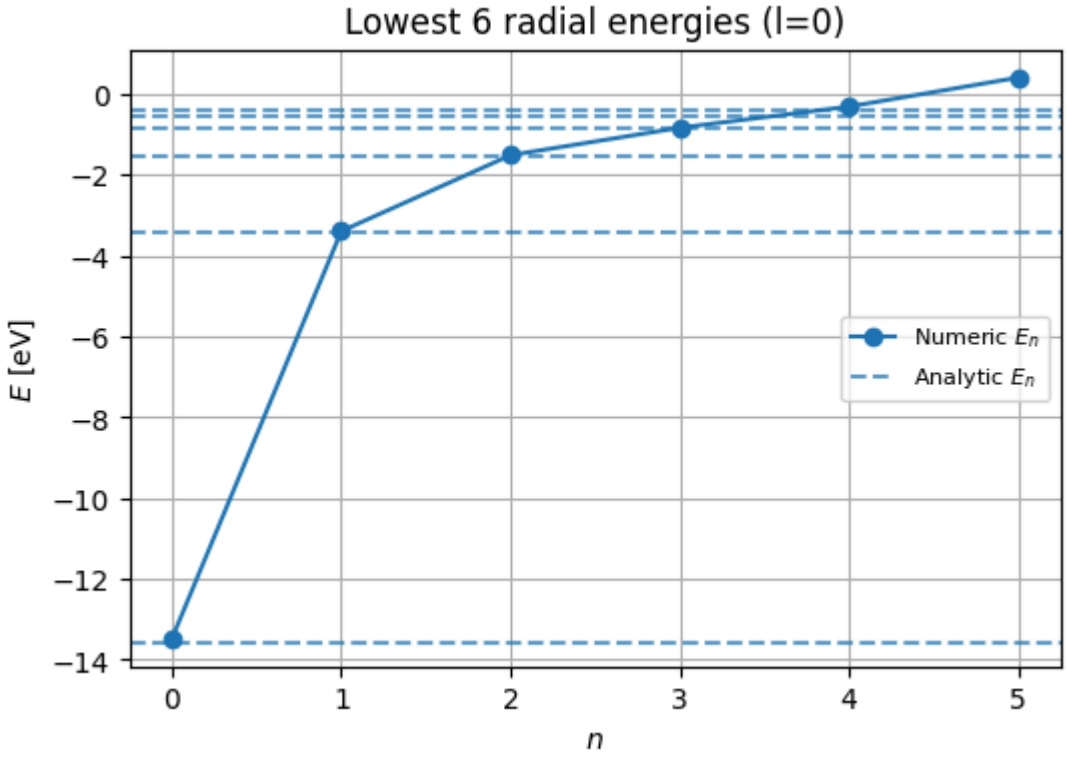
❓ Answer (start)

In [15]:
```
#Operators
L = Laplacian(r)          # d^2/dr^2
l = 0                     # first we compare the wave functions with l=0
Veff = V(r, l)

#Full radial Hamiltonian: H = -1/2 d^2/dr^2 + V(r,l)
H = -0.5 * L + Veff

#Diagonalize
e, v = np.linalg.eigh(H)
```

In [16]:
```
#Convert to eV
e_eV = e * 27.2

print("Lowest 6 energies (in eV):")
print(e_eV[:6])

#Plot the lowest 6 energies
plt.figure(figsize=(6,4))
plt.plot(e_eV[:6], 'o-', label='Numeric $E_n$')
plt.xlabel("$n$")
plt.ylabel("$E$ [eV]")
plt.title("Lowest 6 radial energies (l=0)")
```

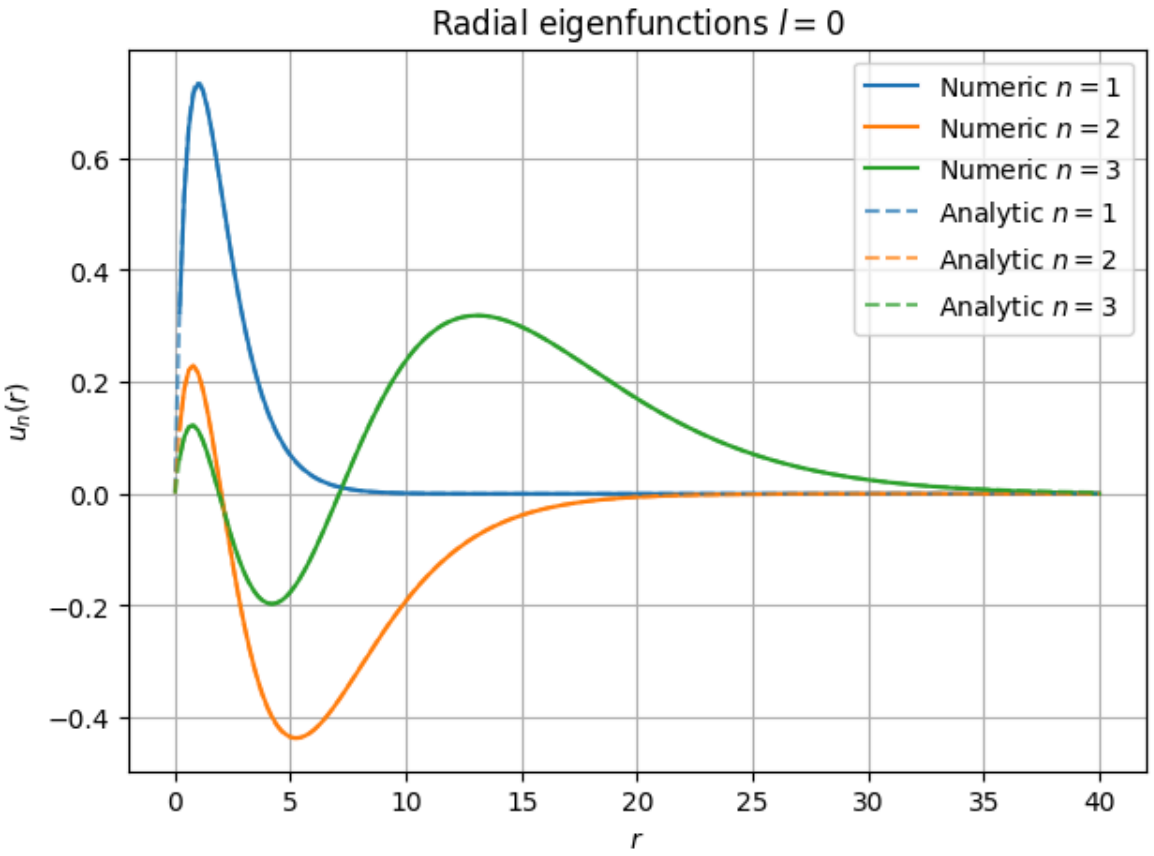
```
#Add analytic reference: E_n = -13.6 / n^2
for n in range(1, 7):   # n=1..5
    En_analytic = -13.6 / n**2
    plt.axhline(En_analytic, linestyle='--', alpha=0.7,
                label="Analytic $E_n$" if n <= 1 else None)

plt.legend(loc='best', fontsize=8)
plt.grid(True)
plt.show()
```

```
Lowest 6 energies (in eV):
[-13.46665385  -3.39154224  -1.50940275  -0.83131389  -0.30682149
   0.41147282]
```



In [17]:
```
dr = r[1] - r[0]
h = abs(dr)      #spacing for normalization

#Make r ascending for plotting
r_plot = r[::-1]

#Take the first three eigenvectors and normalize for plotting
u1_num = (v[:, 0] / np.sqrt(h))[::-1]   # reverse to match r_plot
u2_num = -(v[:, 1] / np.sqrt(h))[::-1]  #(-1) phase factor introduced to match t
u3_num = (v[:, 2] / np.sqrt(h))[::-1]

# Analytic radial u_n(r) = r * R_{n0}(r) for comparison
r_analytic = np.linspace(0.001, 40, 1000)
u1_analytic = r_analytic * radial_function(1, 0, r_analytic)
u2_analytic = r_analytic * radial_function(2, 0, r_analytic)
u3_analytic = r_analytic * radial_function(3, 0, r_analytic)

plt.figure(figsize=(7,5))

#numeric eigenstates
plt.plot(r_plot, u1_num, label="Numeric $n=1$", color='C0')
plt.plot(r_plot, u2_num, label="Numeric $n=2$", color='C1')
```

```
plt.plot(r_plot, u3_num, label="Numeric $n=3$", color='C2')


#analytic curves
plt.plot(r_analytic, u1_analytic, '--', label="Analytic $n=1$", color='C0', alph
plt.plot(r_analytic, u2_analytic, '--', label="Analytic $n=2$", color='C1', alph
plt.plot(r_analytic, u3_analytic, '--', label="Analytic $n=3$", color='C2', alph

plt.xlabel("$r$")
plt.ylabel("$u_n(r)$")
plt.title("Radial eigenfunctions $l=0$")
plt.legend()
plt.grid(True)
plt.show()
```



✓ Answer (end)

d. Finer Grids

We are not getting an accuracy as high as we might desire. What we need is a better grid: finer and going out to larger r. Unfortunately, if there are more grid points, the matrix gets larger and using dense matrices and solving for all the eigenstates becomes intractable. Instead we need to work with sparse matrices.

To do this, you need to do the following:

- Change `V` and `Laplacian` to return sparse matrices. You can do this by just having the return be `return  scipy.sparse.csr_matrix(oldMatrix)`
- When you diagonalize, take `e,v=scipy.sparse.linalg.eigsh(H,k=6,which='SA')` This will give you

sparse versions of the lowest six eigenstates.

Use now a grid of $N = 4000$ out to $L = 60$

Go ahead and plot the same curves as above and verify you get more reasonable results.

? Answer (start)

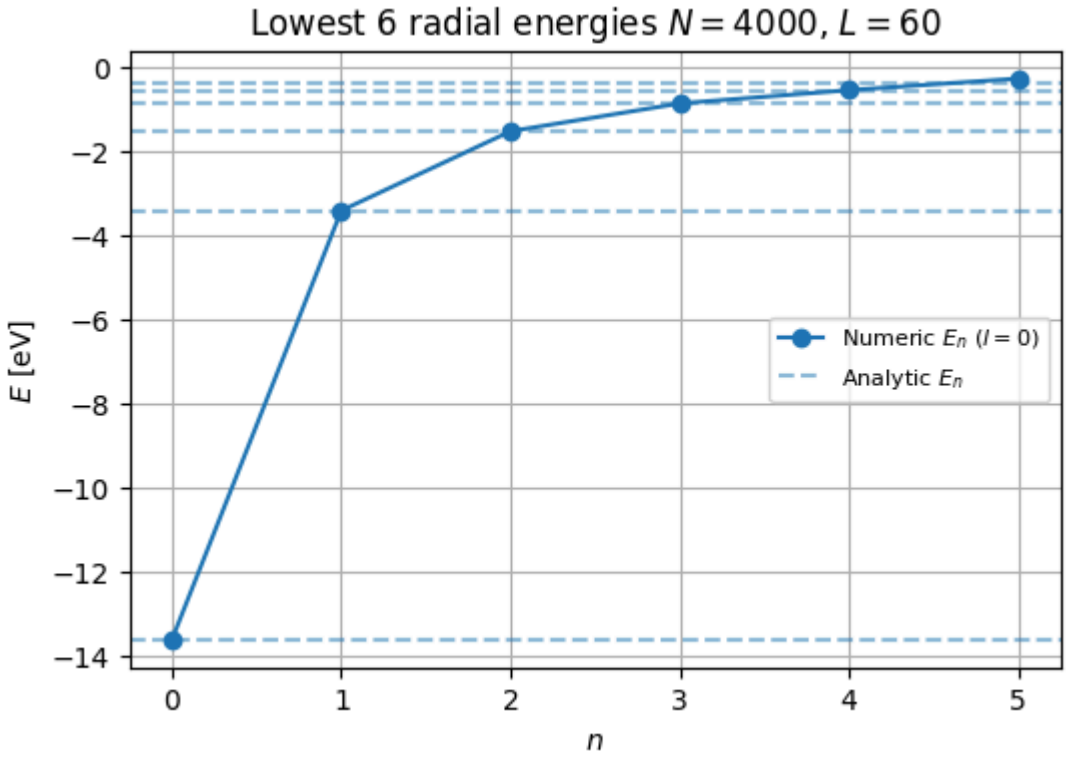
In [18]:
```
#Redefine Laplacian(r) and V(r,l) to return sparse matrices
def Laplacian(r):
    N = len(r)
    dr = r[1] - r[0]
    coeff = 1.0 / (dr**2)

    main_diag = -2.0 * coeff * np.ones(N)
    off_diag  =  1.0 * coeff * np.ones(N - 1)

    #Build complete matrix then convert to sparse
    L_complete = (
        np.diag(main_diag, k=0) +
        np.diag(off_diag,  k=1) +
        np.diag(off_diag,  k=-1)
    )
    return scipy.sparse.csr_matrix(L_complete)

def V(r, l):
    V_values = l*(l+1)/(2 * r**2) - 1.0/r
    V_dense = np.diag(V_values)
    return scipy.sparse.csr_matrix(V_dense)
```

In [19]:
```
#Set up new grid
N = 4000
L = 60.0
r = np.linspace(L, 0.0, N, endpoint=False)
dr = r[1] - r[0]
h = abs(dr)

#Build sparse Hamiltonian
l = 0
L_sp = Laplacian(r)
V_sp = V(r, l)
H = -0.5 * L_sp + V_sp

#Compute the 6 lowest eigenvalues/eigenvectors
e, v = scipy.sparse.linalg.eigsh(H, k=6, which='SA')
```

In [20]:
```
#Plot numeric vs analytic energies

e_eV = e * 27.2

print("Lowest 6 energies (eV):")
print(e_eV)

plt.figure(figsize=(6,4))
plt.plot(e_eV, 'o-', label='Numeric $E_n$ ($l=0$)')
plt.xlabel("$n$")
plt.ylabel("$E$ [eV]")
plt.title("Lowest 6 radial energies $N=4000$, $L=60$")
```

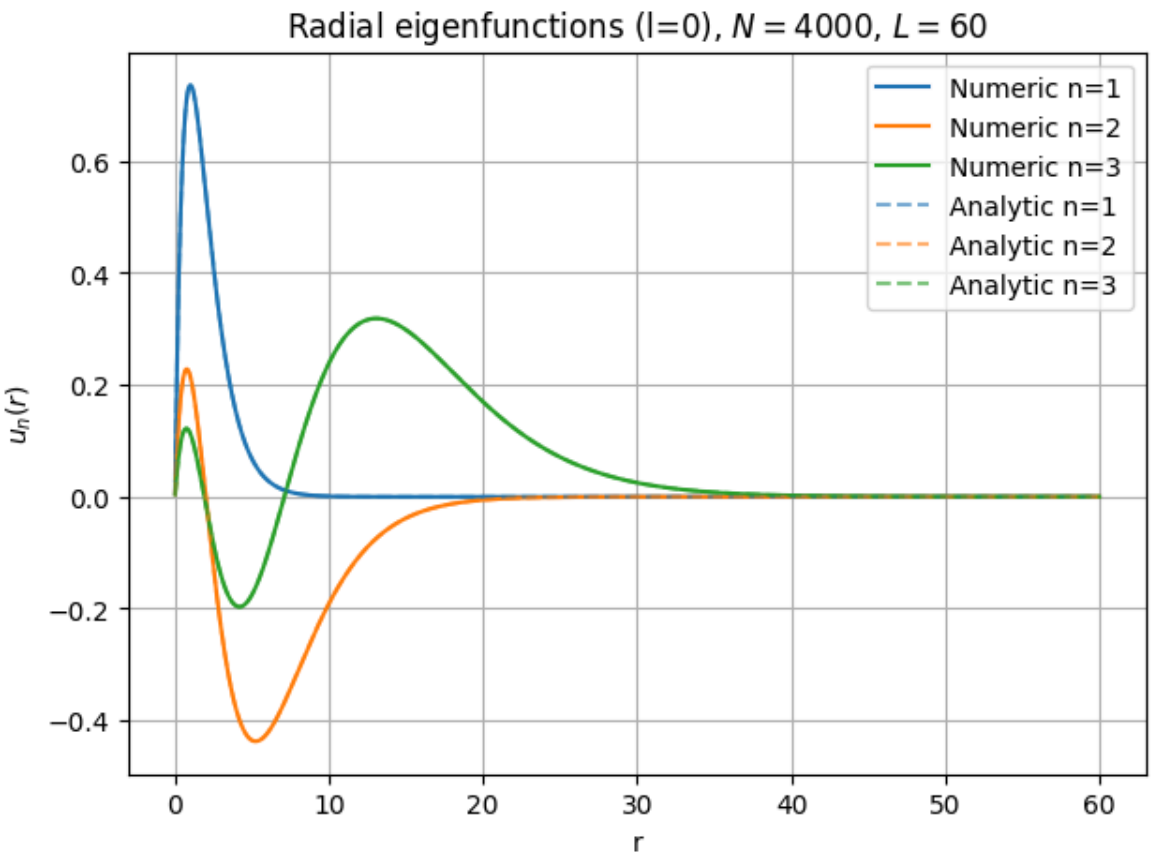
```
for n in range(1, 7):
    En_analytic = -13.6 / n**2
    plt.axhline(En_analytic, linestyle='--', alpha=0.5,
                label=f"Analytic $E_n$" if n <= 1 else None)

plt.legend(loc='best', fontsize=8)
plt.grid(True)
plt.show()
```

```
Lowest 6 energies (eV):
[-13.59923509  -3.39995219  -1.51110167  -0.84994694  -0.53305305
  -0.25801611]
```



In [21]:
```
#Plot numeric vs analytic radial functions

#Function to align phase of numerical solution, given that the analytical sol. i
def align_phase(u_num):
    if u_num[5] >= 0:   #take random value close to 0
        return u_num    #nothing to do
    phase = (-1)        #the eigenvectors after being normalized can be out of p
    u_aligned = u_num * np.conjugate(phase)
    return u_aligned

# Reverse r
r_plot = r[::-1]

# Take first three eigenvectors and normalize
u1_num = (v[:, 0] / np.sqrt(h))[::-1]
u2_num = (v[:, 1] / np.sqrt(h))[::-1]
u3_num = (v[:, 2] / np.sqrt(h))[::-1]


# Analytic u_n(r) = r * R_{n0}(r)
r_analytic = np.linspace(0.001, L, 2000)
u1_analytic = r_analytic * radial_function(1, 0, r_analytic)
u2_analytic = r_analytic * radial_function(2, 0, r_analytic)
```

```
u3_analytic = r_analytic * radial_function(3, 0, r_analytic)

# Aligning phases
u1_num = align_phase(u1_num)
u2_num = align_phase(u2_num)
u3_num = align_phase(u3_num)

#Plots
plt.figure(figsize=(7,5))

plt.plot(r_plot, u1_num, label="Numeric n=1", color='C0')
plt.plot(r_plot, u2_num, label="Numeric n=2", color='C1')
plt.plot(r_plot, u3_num, label="Numeric n=3", color='C2')

plt.plot(r_analytic, u1_analytic, '--', label="Analytic n=1", color='C0', alpha=
plt.plot(r_analytic, u2_analytic, '--', label="Analytic n=2", color='C1', alpha=
plt.plot(r_analytic, u3_analytic, '--', label="Analytic n=3", color='C2', alpha=

plt.xlabel("r")
plt.ylabel("$u_n(r)$")
plt.title("Radial eigenfunctions (l=0), $N=4000$, $L=60$")
plt.legend()
plt.grid(True)
plt.show()
```



✓ Answer (end)

Exercise 3. Hydrogen Atom on a Grid

So far we've plotted the orbitals for the Hydrogen atom by

- using the formula for the orbitals that one derives analytically
- using the radial equation you derived analytically (along with the spherical harmonics)

In both cases, we (somebody?) had to do a lot of analytic work to get something that we then worked with computationally. Now we just want to envision a situation where we have a Hamiltonian and want to directly work with and diagonalize this on a grid.

a. The Hamiltonian

Our goal here is to be able to solve for the eigenstates and eigenvalues of the Hydrogen atom.

The hydrogen atom has a single electron and a Hamiltonian of

$$H = -\frac{1}{2}\nabla^2 + V(R)$$

where

$$V(R) = -\frac{1}{R}$$

where R is the distance between the proton and electron of the Hydrogen atom.

Let's learn some things to put this together:

First recall that $\nabla^2 = \frac{d^2}{dx^2} + \frac{d^2}{dy^2} + \frac{d^2}{dz^2}$. We need to generate a matrix for this laplacian.

There's something a little subtle here. d^2/dx^2 looks like the same term that we saw for the particle in the box but it now needs to exist over three-dimensions. If L was the one-dimensional derivative, we instead now write this as $L \otimes I \otimes I$ where I is the identity of the same size matrix.

Let's think how large a matrix we are making here. We have a matrix that is number of grid points $\times$ grid points. If we use 80 grid points in every direction, we get a matrix of size $80^3 \times 80^3$ or $512,000 \times 512,000$. This is much too large to be densely represented on a computer. To get around this, we are going to have to use sparse matrices. The trick here is to

- first take your one-dimensional Laplacian (call it `L`) and turn it into a sparse matrix (i.e. `scipy.sparse.lil_matrix(L)`) and
- then when you produce your three-dimensional laplacian make sure that you do a kroneker product - i.e. `scipy.sparse.kron(scipy.sparse.kron(L,I),I)` These two steps will give you $\frac{d^2}{dx^2}$. Then you need to also get $\frac{d^2}{dy^2}$ and $\frac{d^2}{dz^2}$. Go ahead and generate the matrix then for ∇^2 (make sure in your laplacian you rememember to divide by $1/h^2$ where h is the grid spacing).

Let's check that our Laplacian is giving us the correct thing. We should be able to diagonalize it and get the energies for a particle in a three-dimensional box.

You can't actually afford to get all the eigenvalues of such a large matrix. Instead, we are going to only get the first couple. To do this, you can use `e,v=scipy.sparse.linalg.eigsh(-0.5*L, which='SA')` and show that the first few energies are correct for a grid

```
HydrogenGrid(12,numGridPoints=80)
```

The `which='SA'` is important to get the lowest energies.

? Answer (start)

In [22]:
```
#Set up grid
x, y, z, h = HydrogenGrid(12, numGridPoints=80)
N = x.shape[0]   # N = 80 points per direction

#One dimensional Laplacian
def Laplacian1D(N, h):
    coeff = 1.0 / (h**2)
    main_diag = -2.0 *coeff* np.ones(N)
    off_diag  =  1.0 *coeff* np.ones(N - 1)

    L_complete = (
        np.diag(main_diag, k=0) +
        np.diag(off_diag,  k=1) +
        np.diag(off_diag,  k=-1)
    )

    # Convert to sparse lil_matrix as requested
    L_sparse = scipy.sparse.lil_matrix(L_complete)
    return L_sparse

#3D Laplacian using kronecker products

def Laplacian3D(N, h):
    L1D = Laplacian1D(N, h)                          # d^2/dx^2 in 1D
    I   = scipy.sparse.identity(N, format='lil')   # 1D identity

    #Build the three directions via Kronecker products
    Lx = scipy.sparse.kron(scipy.sparse.kron(L1D, I), I)
    Ly = scipy.sparse.kron(scipy.sparse.kron(I, L1D), I)
    Lz = scipy.sparse.kron(scipy.sparse.kron(I, I), L1D)

    L3D = Lx + Ly + Lz

    return L3D
```

In [23]:
```
#First two eigenvalues/eigenstates

L3D = Laplacian3D(N, h)

#Kinetic Energy (Hamiltonian of a 3D box):
H_box = -0.5 * L3D

#Diagonalizing H_box
e, v = scipy.sparse.linalg.eigsh(H_box, k=2, which='SA')
e = np.sort(e)   # just to be sure they're in ascending order
```

```
print("Lowest 2 numerical energies for particle in 3D box:")
print(e)
```

```
Lowest 2 numerical energies for particle in 3D box:
[0.02444546 0.04887866]
```

In [24]:
```
#Analytic energies for a 3D box of length L=2g

# Analytic energies for a 3D box of length L = 2g
L = 24  # For HydrogenGrid(12)

print("Lowest 2 analytical energies for particle in 3D box:")

def E(i, j, k):
    return (np.pi**2 / (2 * L**2)) * (i**2 + j**2 + k**2)

for (i, j, k) in [(1, 1, 1), (2, 1, 1)]:
    print(f"n=({i},{j},{k})  →  E = {E(i, j, k)}")
```

```
Lowest 2 analytical energies for particle in 3D box:
n=(1,1,1)  →  E = 0.025702094794503538
n=(2,1,1)  →  E = 0.051404189589007075
```

✓ Answer (end)

After you have your laplacian in place, you want to generate the Coulomb potential piece of your Hamiltonian ($1/R$) on a grid.

The potential $V(R)$ is a diagonal term of the Hamiltonian (i.e. it's on the diagonal of the matrix). First compute the potential $V(R)$ on your three-dimensional grid (call it `Vext`). To put it on the diagonal, you need to reshape your three-dimensional grid onto a one-dimensional vector which you can then make the diagonal of your matrix

```
m=np.shape(Vext)[0]
Vext=np.reshape(Vext,m*m*m)
potential=scipy.sparse.diags(Vext)
```

Write a function `ExternalPotential` which generates this matrix.

You now have your laplacian term and your potential term. You can add these two terms together to get your full Hamiltonian.

? Answer (start)

In [25]:
```
def ExternalPotential(x, y, z):
    # 3D radius
    R = np.sqrt(x**2 + y**2 + z**2)
    # Coulomb potential V(R) = -1/R, avoid division by zero at R=0
    Vext = np.zeros_like(R)
    nonzero = (R != 0)
    Vext[nonzero] = -1.0 / R[nonzero]
    # At R=0, leave Vext = 0
    #3D grid to 1D vector
    m = Vext.shape[0]          #cubic grid m * m * m
    Vflat = np.reshape(Vext, m*m*m)
    # Make sparse diagonal matrix
```

```
    Vop = scipy.sparse.diags(Vflat)
    return Vop
```

In [26]:
```
#Computing the coulomb potential for the grid
Vop = ExternalPotential(x, y, z)

#Full hamiltonian
H = -0.5 * L3D + Vop
```

✓ Answer (end)

b. Eigenvalues

Diagonalize your full Hamiltonian using `scipy.sparse.linalg.eigsh` on your $12 \times 12 \times 12$ grid.

We would like to see that you get the expected eigenenergies and eigenstates.

Recall the standard hydrogen orbitals from chemistry each with an energy of $-13.6/n^2$ eV

The lowest 6 energies for the Hydrogen atom should be

- 1s
- 2s
- 2px, 2py, 2pz
- 3s

Let's start with looking at the energies of the 1s, 2s, and 3s states which we expect it to be -13.6 eV, -13.6/4 eV and -13.6/9 eV respectively. Obviously because we are using a grid, we aren't going to get the exact answer. But as we increase the grid size we should get closer and closer to the true answer. Starting with 20 grid points per direction and increasing the grid in units of 20 up to 100 grid points per direction, graph the energy of these three eigenstates as a function of number of grid points per direction. Also draw horizontal lines at the expected value of the eigenstates. You should see the energies approach the target energies at large grid-size. *Hint: Your g=20 energy for 1s should be -10.71473301*. (There are some eigenstates where you need to increase the size of the grid to capture the full wave-function)

Note:. You need to be careful about units. To convert into eV you have to multiply your energy by 27.21.

? Answer (start)

In [27]:
```
#Diagonalize the Hamiltonian from part (a)

#Get the lowest 6 eigenvalues/eigenvectors
e, v = scipy.sparse.linalg.eigsh(H, k=6, which='SA')

#Sort them
idx = np.argsort(e)
e = e[idx]
```

```
v = v[:, idx]

e_eV = e * 27.21

print("Lowest 6 energies (eV) for N=80, g=12:")
print(e_eV)
```

```
Lowest 6 energies (eV) for N=80, g=12:
[-13.31463754  -3.3974114   -3.3974114   -3.3974114   -3.34586968
  -1.25351682]
```

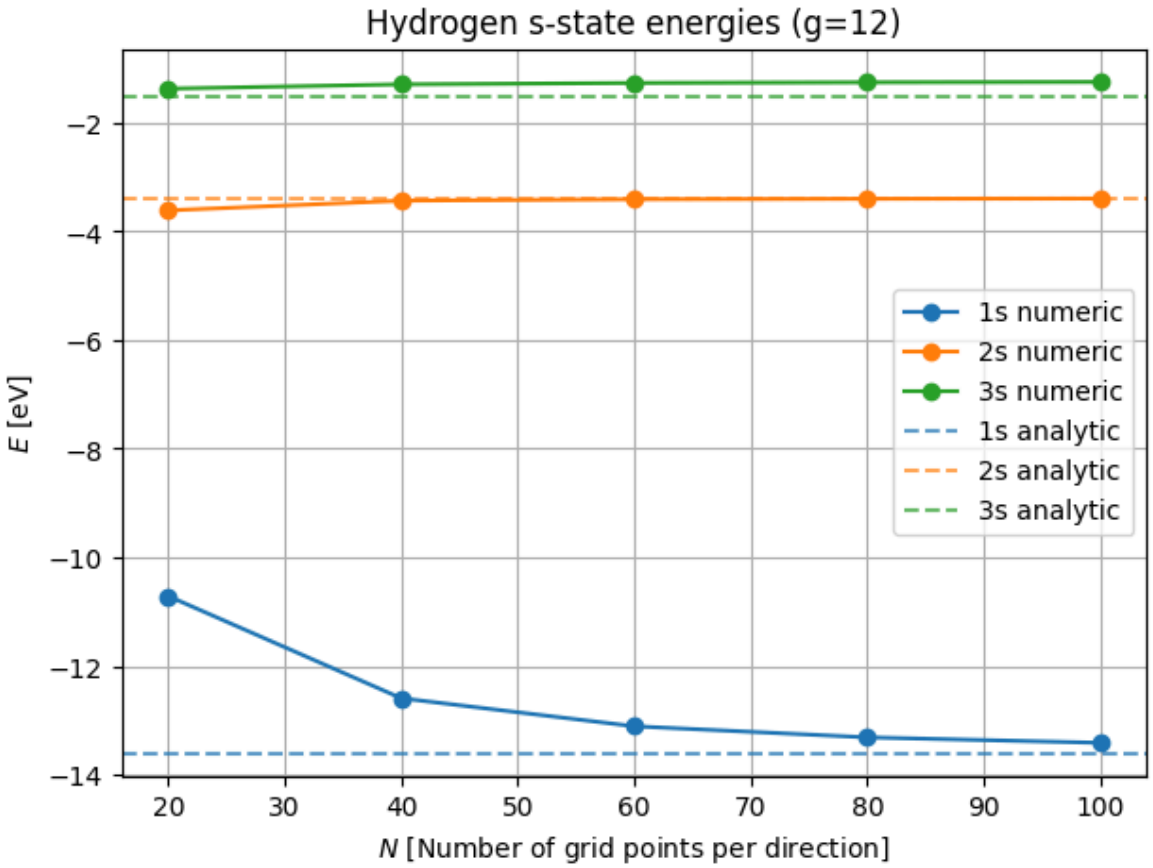
In [28]:
```
#Function to build the 3D Hydrogen hamiltonian using the parameters g and N
def hydrogen_H(g, N):
    x, y, z, h = HydrogenGrid(g, numGridPoints=N)
    L3D = Laplacian3D(N, h)
    Vop = ExternalPotential(x, y, z)
    H = -0.5 * L3D + Vop
    return H, x, y, z, h

H, x, y, z, h = hydrogen_H(12, 20)

#lowest 6 eigenvalues
e, v = scipy.sparse.linalg.eigsh(H, k=1, which='SA')
e_eV = e * 27.21        # to eV
print("S1 energy for N=20, g=12:", e_eV)
```

```
S1 energy for N=20, g=12: [-10.71473184]
```

In [29]:
```
#1s, 2s, 3s stateswith g=20 and grid points ranging from 20 to 100

#Diagonalizing according to the grid sizes

grid_sizes = [20, 40, 60, 80, 100]
g = 12

E1s = []
E2s = []
E3s = []

for N in grid_sizes:
    print(f"N={N} ...")
    H, x, y, z, h = hydrogen_H(g, N)

    #lowest 6 eigenvalues
    e, v = scipy.sparse.linalg.eigsh(H, k=6, which='SA')
    idx = np.argsort(e)
    e = e[idx]              #sorted in ascending order

    e_eV = e * 27.21        # to eV

    # pick 1s, 2s, 3s
    E1s.append(e_eV[0])     # ground state
    E2s.append(e_eV[1])     # 2s-like state
    E3s.append(e_eV[5])     # 3s-like state

    print(f"  1s ≈ {e_eV[0]: .6f} eV")
    print(f"  2s ≈ {e_eV[1]: .6f} eV")
    print(f"  3s ≈ {e_eV[5]: .6f} eV")
    print()
```

```
N=20 ...
  1s ≈ -10.714732 eV
  2s ≈ -3.611558 eV
  3s ≈ -1.370241 eV

N=40 ...
  1s ≈ -12.594718 eV
  2s ≈ -3.432229 eV
  3s ≈ -1.290332 eV

N=60 ...
  1s ≈ -13.110340 eV
  2s ≈ -3.406254 eV
  3s ≈ -1.265620 eV

N=80 ...
  1s ≈ -13.314638 eV
  2s ≈ -3.397411 eV
  3s ≈ -1.253517 eV

N=100 ...
  1s ≈ -13.414799 eV
  2s ≈ -3.393261 eV
  3s ≈ -1.246332 eV
```

In [30]:
```python
#Plot the numeric and analytic solutions to be compared
plt.figure(figsize=(7,5))

plt.plot(grid_sizes, E1s, 'o-', label='1s numeric')
plt.plot(grid_sizes, E2s, 'o-', label='2s numeric')
plt.plot(grid_sizes, E3s, 'o-', label='3s numeric')

# Horizontal analytic lines
E1s_analytic = -13.6
E2s_analytic = -13.6 / 4.0
E3s_analytic = -13.6 / 9.0

plt.axhline(E1s_analytic, color='C0', linestyle='--', alpha=0.7, label='1s analy
plt.axhline(E2s_analytic, color='C1', linestyle='--', alpha=0.7, label='2s analy
plt.axhline(E3s_analytic, color='C2', linestyle='--', alpha=0.7, label='3s analy

plt.xlabel("$N$ [Number of grid points per direction]")
plt.ylabel("$E$ [eV]")
plt.title("Hydrogen s-state energies (g=12)")
plt.legend()
plt.grid(True)
plt.show()
```



✓ Answer (end)

c. Eigenvectors

Now we can also look at the eigenstates. For this section, let's use a grid with 80 points.

Let's start by looking at the s-states. First, we will make some two-dimensional slices of the three-dimensional wave-functions we've found. To get your 2d visualization, you first need to reshape your wave-function into a 3d grid - if you have a $80 \times 80 \times 80$ grid this can be done for the i th eigenstate by doing `A=psi[:,i].reshape(80,80,80)`. Now you can go ahead and plot it by `plt.matshow(A[40,:,:])` where 40 is the slice that you want. You probably want to use `plt.colorbar()` so you can see the magnitude and sign of things.

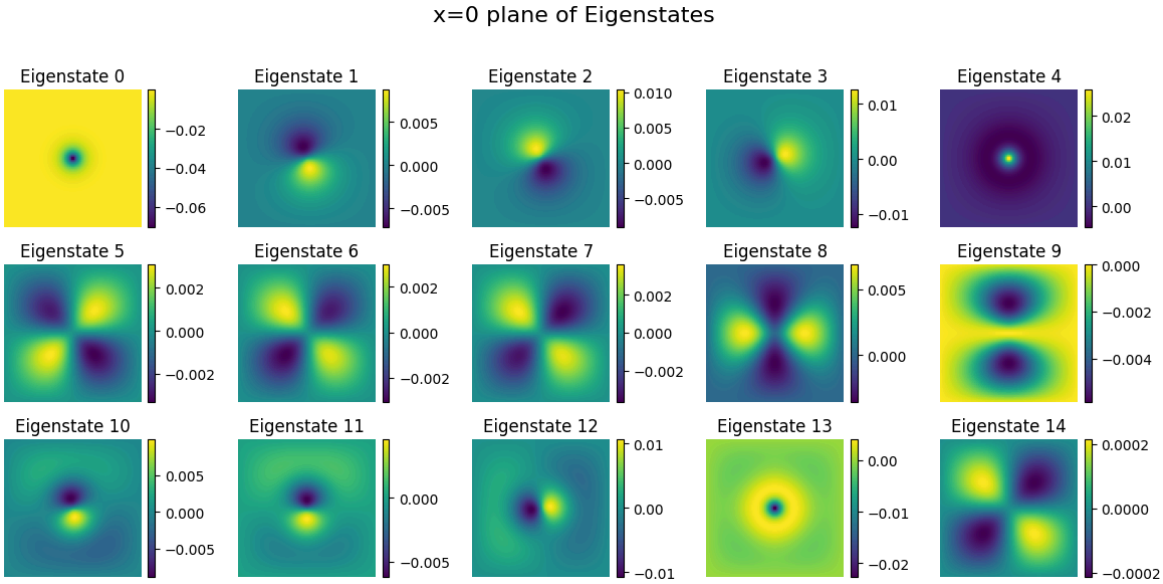
If you look at the 0 (1s),1 (2s) or 5 (3s) eigenstate you will find that it should have a clear "S" character - symmetric around the origin. (This ordering of eigentstates)

You can also graph the probability density as a function of the radius. You could average but because it is symmetric the easiest thing to do is just plot a one dimensional line starting at the origin. To match up with earlier results, actually we'd like to plot not the radial function $|R_{nl}(r)|^2$ but $|u_{nl}(r)|^2$ so multiply by r - i.e. `plt.plot(r,r*A[40,40,40:])`. Notice we are starting at `40:` for that last term because that's where zero starts. For the different shells, count how many nodes you see (i.e. places where the probability density goes to zero). To normalize you have to divide by $1/(4\pi\sqrt{h})$

? Answer (start)

In [31]:
```
#Getting the lowest 15 energy states
g=12
N=80
H, x, y, z, h = hydrogen_H(g, N)
e, v = scipy.sparse.linalg.eigsh(H, k=15, which='SA')
```

In [32]:
```
#Rename v as psi
psi = v

num_states = psi.shape[1]

#Plot eigenstates 0 through 14 (or fewer if you computed less)
max_plot = min(15, num_states)

slice_index = N//2   #Center slice

fig, axes = plt.subplots(3, 5, figsize=(12, 6))
axes = axes.flatten()

for i in range(max_plot):
    A = psi[:, i].reshape(N, N, N)
    ax = axes[i]

    im = ax.imshow(np.real(A[slice_index, :, :]), cmap='viridis')
    ax.set_title(f"Eigenstate {i}")
    ax.axis('off')
    fig.colorbar(im, ax=ax, fraction=0.046, pad=0.04)

for j in range(max_plot, 15):
    axes[j].axis('off')

plt.suptitle("x=0 plane of Eigenstates", fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```



Here the eigenstates 0,4,13 correspond to 1s,2s,3s. Furthermore in this plot we can already see the states with p-character asked below.

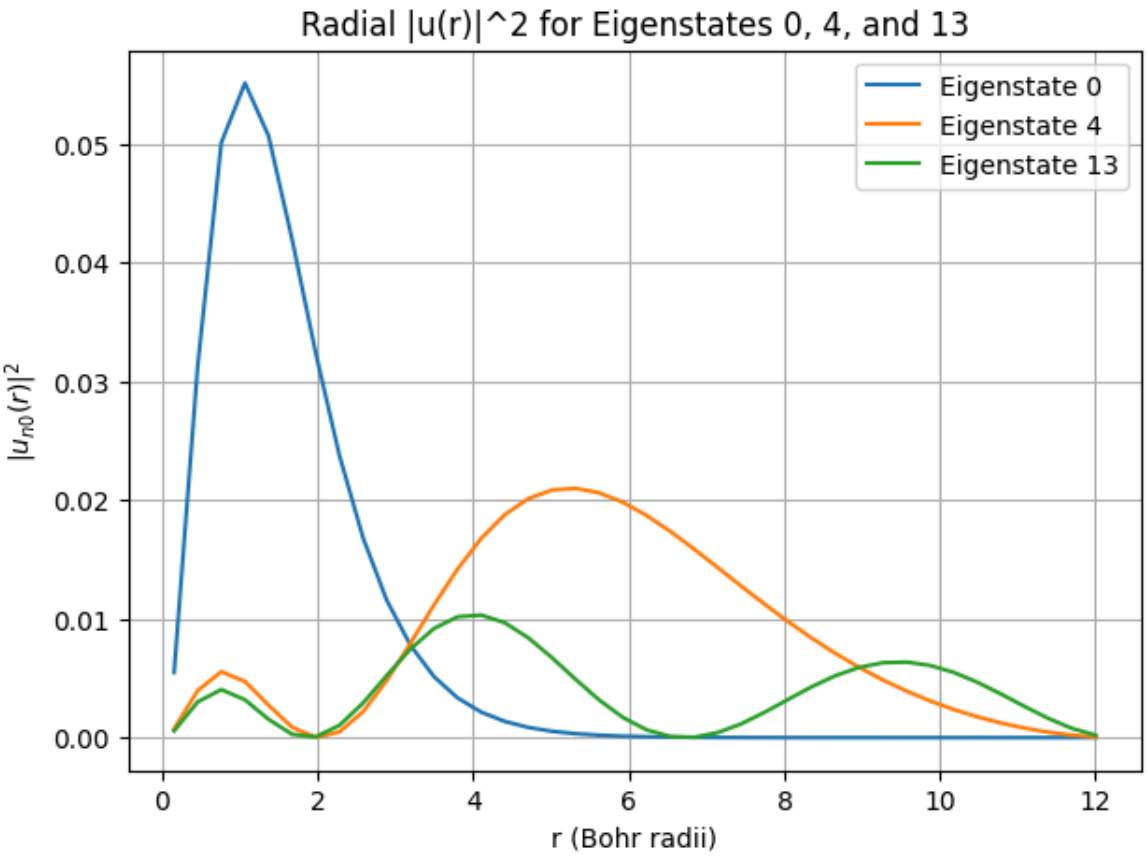
✓ Answer (end)

You can also look at the 2'nd eigenstate (and third and fourth) eigenstate on a 2d plot. This should show clearly the p-character of your state.

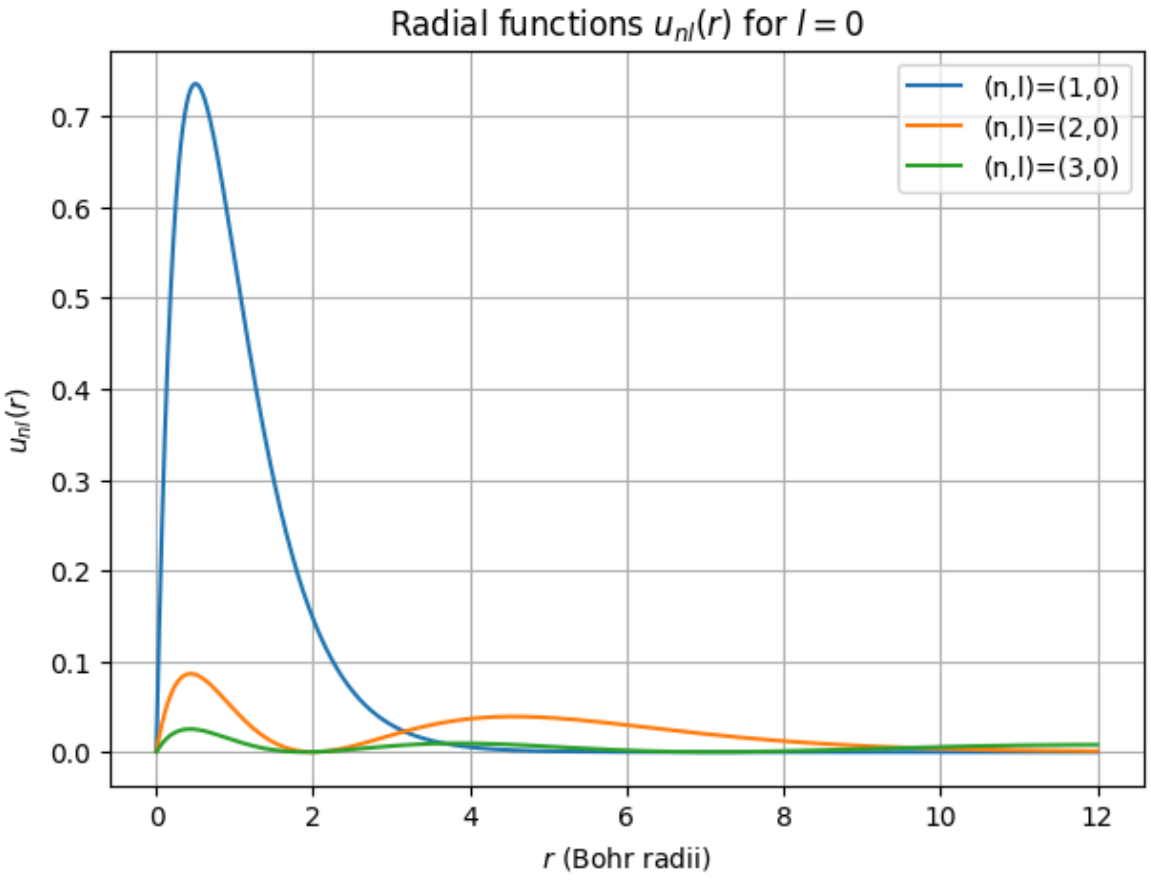
? Answer (start)

Here we plot the radial equation and compare it to the analytic solution

In [33]:
```
states_to_plot = [0, 4, 13]          # the eigenstate indices
slice_index = N // 2                  # central slice index (x=y=0 axis)
p = np.linspace(-g, g, N)             # coordinate grid
r_line = p[slice_index:]              # r ≥ 0 along +z

norm_factor = 4.0 * np.pi * np.sqrt(h)

plt.figure(figsize=(7,5))

for i in states_to_plot:
    A = psi[:, i].reshape(N, N, N)              # reshape eigenstate i
    psi_line = A[slice_index, slice_index, slice_index:]   # line along +z

    u_line = r_line * psi_line * norm_factor
    P_line = np.abs(u_line)**2

    plt.plot(r_line, P_line, label=f"Eigenstate {i}")

plt.xlabel("r (Bohr radii)")
plt.ylabel(r"$|u_{n0}(r)|^2$")
plt.title("Radial |u(r)|^2 for Eigenstates 0, 4, and 13")
plt.grid(True)
plt.legend()
plt.show()
```



In [34]:
```
r = np.linspace(0.001, 12, 1000)

#Plot u_{n l}(r) = r * R_{n l}(r)

plt.figure(figsize=(7,5))

for n in [1, 2, 3]:
    l = 0
    R = np.abs(radial_function(n, l, r))**2
    u = r * R
    plt.plot(r, u, label=f"(n,l)=({n},{l})")

plt.xlabel("$r$ (Bohr radii)")
plt.ylabel("$u_{nl}(r)$")
plt.title("Radial functions $u_{nl}(r)$ for $l=0$")
plt.legend()
plt.grid(True)
plt.show()
```



✓ Answer (end)

c. Three-dimensional plots

Now we'd like to make some three-dimensional plots.

Make some three-dimensional plots. To do this, you need to put your wave-function back on a grid - i.e. psi[:,i].reshape(80,80,80).

Then use the following function

```
def PlotMe(psi):
    isomin=np.min(np.abs(psi)**2)
    isomax=np.max(np.abs(psi)**2)
    print(isomin,isomax)
    g=80
    p = np.linspace(-12, 12, g);         #% one dimensiton space
lattice
    h = p[2] - p[1]              #% latice spacing
    L3=Laplacian(h,g)
    [X, Y, Z] = np.meshgrid(p, p, p);  #% three d
    # Create the isosurface
    isosurface = go.Isosurface(
        x=X.flatten(),
        y=Y.flatten(),
        z=Z.flatten(),
        value=(np.abs(psi)**2).flatten(),  # Use the absolute
value for intensity
        #isomin=0.01**2,  # Adjust the minimum intensity
```

```
    threshold
            #isomax=isomax,  # Adjust the maximum intensity
    threshold
            surface_count=20, # Number of isosurfaces to display
            #colorscale='Viridis',  # Choose a color scale
            opacity=0.5,  # Set the opacity of the isosurface
            surface_fill=1.0,
            caps=dict(x_show=False, y_show=False)
        )
        fig = go.Figure(data=[isosurface])
        fig.show()
```

Compare these orbitals to the ones you got by directly plotting the analytic formula.

? Answer (start)

In [37]:
```
def PlotMe(psi):
    isomin = np.min(np.abs(psi)**2)
    isomax = np.max(np.abs(psi)**2)
    print(isomin, isomax)

    g = 80
    p = np.linspace(-12, 12, g)
    h = p[1] - p[0]

    X, Y, Z = np.meshgrid(p, p, p, indexing='ij')

    isosurface = go.Isosurface(
        x=X.flatten(),
        y=Y.flatten(),
        z=Z.flatten(),
        value=(np.abs(psi)**2).flatten(),
        surface_count=20,
        opacity=0.5,
        surface_fill=1.0,
        caps=dict(x_show=False, y_show=False, z_show=False)
    )

    fig = go.Figure(data=[isosurface])
    fig.update_layout(title='3D Hydrogen Numerical Solutions')
    fig.show()


#Plot the numerical orbitals

states_to_plot = [0, 3, 7, 8]

for i in states_to_plot:
    print(f"\nEigenstate {i} ...")
    A = psi[:, i].reshape(80, 80, 80)
    PlotMe(A)
```

```
Eigenstate 0 ...
1.0436507861704976e-23 0.004960080358632725
Eigenstate 3 ...
1.8807583853629292e-20 0.00015684246572920187
Eigenstate 7 ...
2.5643084940370744e-18 3.4497414375513996e-05
```

LOOK AT THE END OF THE DOCUMENT TO FIND THE IMAGES OF THE 3D PLOTS!

```
Eigenstate 8 ...
4.895002461567824e-37 4.7122433383781967e-05
```

✓ Answer (end)

Acknowledgements:

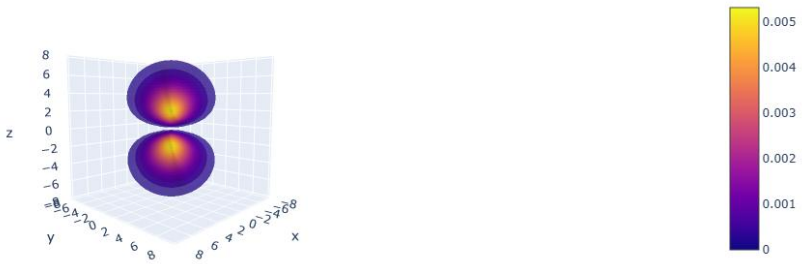
- Bryan Clark (original)

In [ ]:

In [ ]:

Orbital (1,0,0)

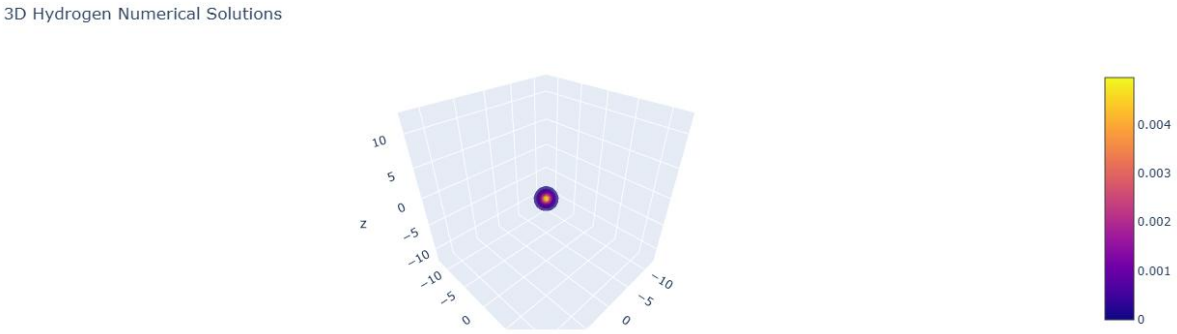
Orbital (2,1,0)



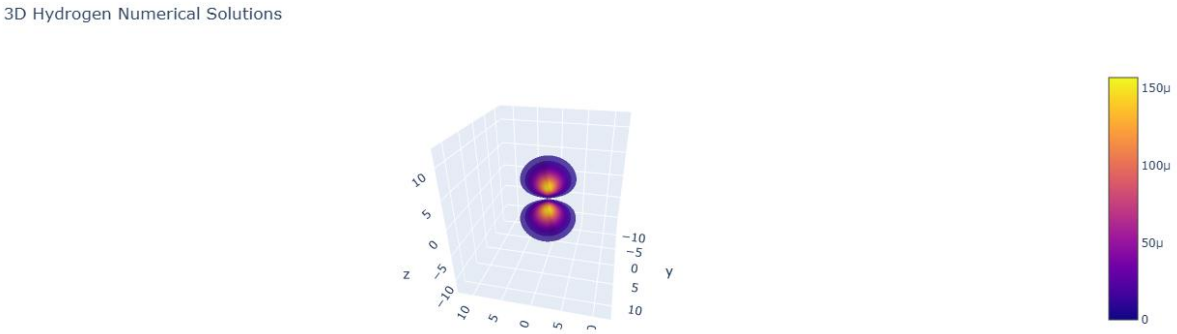
Orbital (3,2,2)

Orbital (3,2,0)

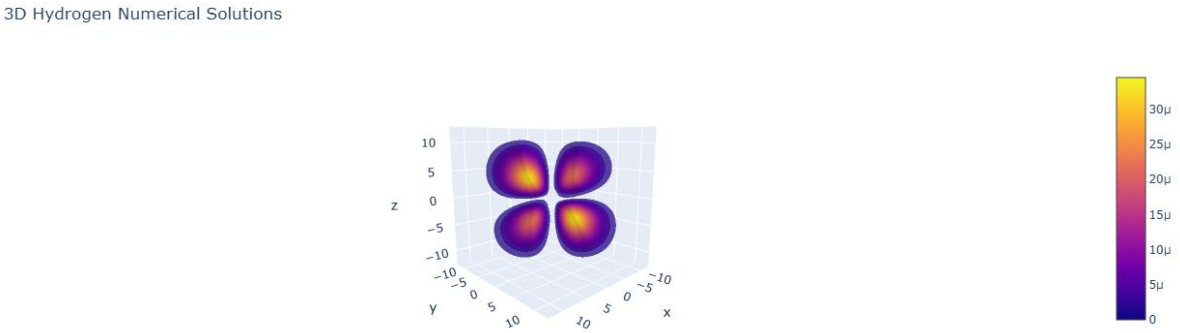
Eigenstate 0 – Orbital (1,0,0)



Eigenstate 3 - (2,1,0)



Eigenstate 7 – (3,2,2)



Eigenstate 8 – (3,2,0)