1 Derivatives

By definition,

$$f'(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}. \tag{1}$$

On a uniform grid $\{x_i\}$, the smallest distance we can represent is the grid spacing, h. A natural choice is therefore the following difference quotient:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{2}$$

But this is a *left* derivative: It uses x and $x+h$ to approximate the derivative at x. Intuition should tell us (right?) that it actually best approximates the derivative *between* those two points, i.e., $f'(x+h/2)$, which is not part of our grid. If we are going to do computations with both f and f', we should wisely try to have them on the *same* grid so we can easily do arithmetic with them. We therefore write:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{3}$$

We say that this is a *central* finite-difference derivative since the function values are symmetric around the point where we calculate the derivative. To get the second-order derivative, we apply this expression twice:

$$\begin{aligned} f''(x) &\approx \frac{1}{2h}\left[f'(x+h) - f'(x-h)\right] \\ &= \frac{1}{4h^2}\left[f(x+2h) - 2f(x) + f(x-2h)\right]. \end{aligned} \tag{4}$$

In this expression we only see differences of $2h$. Hence we take $2h$ to be the grid spacing and rewrite accordingly:

$$f''(x) \approx \frac{1}{h^2}\left[f(x+h) - 2f(x) + f(x-h)\right]. \tag{5}$$

In conclusion, this is how we would calculate the second-order derivative from function values on a grid.

One can use a higher-order Taylor expansion and obtain expressions that involve several other "nearest neighbours": $f(x), f(x \pm h), f(x \pm 2h), \ldots$. Such expressions give higher accuracy if the grid is fine enough.

How can we represent the kinetic operator as a matrix? Note how the expression for the second derivative is a linear combination of function values at different (neighbouring) grid points. We arrange the coefficients -2 and $+1$ in the diagonal and the first off-diagonals. Then it is straightforward to verify

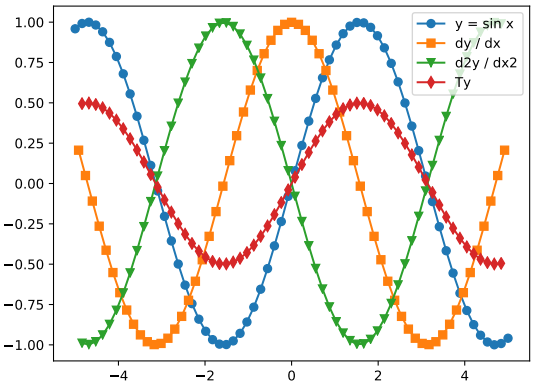

Figure 1: Derivatives calculated by finite differences, and action of the kinetic operator $\hat{T}$.

that:

$$\mathbf{Ty} = -\frac{1}{2h^2}\begin{bmatrix} -2 & 1 & 0 & & \cdots & 0 \\ 1 & -2 & 1 & & & \vdots \\ 0 & 1 & -2 & & & \\ & & & \ddots & & 0 \\ \vdots & & & & -2 & 1 \\ 0 & \cdots & & 0 & 1 & -2 \end{bmatrix}\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_i) \\ \vdots \\ f(x_N) \end{bmatrix}$$

$$= -\frac{1}{2h^2}\begin{bmatrix} \vdots \\ f(x_i - h) - 2f(x_i) + f(x_i + h) \\ \vdots \end{bmatrix}. \tag{6}$$

All the derivatives are shown on Figure 1. The script which calculates and plots them is this:

```
import numpy as np
import matplotlib.pylab as plt

N = 64
x = np.linspace(-5, 5, N)
h = x[1] - x[0]  # Spacing

y = np.sin(x)

plt.plot(x, y, 'o-', label='y = sin x')
```

```
dydx = (y[1:] - y[:N - 1]) / h

# Stencil is most accurate *between* grid points:
xplushalf = 0.5 * (x[1:] + x[:-1])

# Ignore end points of grid as necessary:
d2ydx2 = (y[2:] -2.0 * y[1:-1] + y[:-2]) / h**2

plt.plot(xplushalf, dydx, 's-', label='dy / dx')
plt.plot(x[1:-1], d2ydx2, 'v-', label='d2y / dx2')

T = np.zeros((N, N))
for i in range(N - 1):
    T[i, i] = -2.0
    T[i, i + 1] = 1.0
    T[i + 1, i] = 1.0
T[-1, -1] = -2.0
T *= -0.5 / h**2

Ty = np.dot(T, y)

# Derivative will be discontinuous at the end of the grid
# unless it approaches zero there.  Plot only the interiour:
plt.plot(x[1:-1], Ty[1:-1], 'd-', label='Ty')
print(T)

plt.legend()

plt.savefig('derivatives.pdf')
plt.show()
```

2 Free particles and the harmonic oscillator

If we simply take **T** to be the whole Hamiltonian, we are calculating non-interacting particles within a box as large as our grid. We get the independent-particle wavefunctions using this script: `independent_particles.py` This gives the wavefunctions shown on Figure 2.

The second part of the above listed script adds a quadratic potential to obtain the wavefunctions for the harmonic oscillator, shown on Figure 3.

Finally we need to implement the different potentials and a self-consistency loop.

```
import numpy as np
import matplotlib.pyplot as plt

xmax = 6.0  # Box size
Ng = 200  # Number of grid points
Nn = 3  # Number of states in our calculation
```

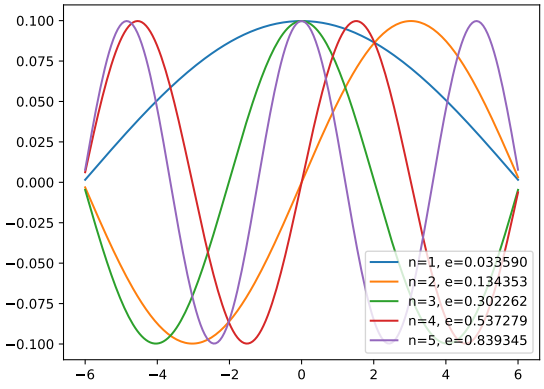

Figure 2: Particles in a box.

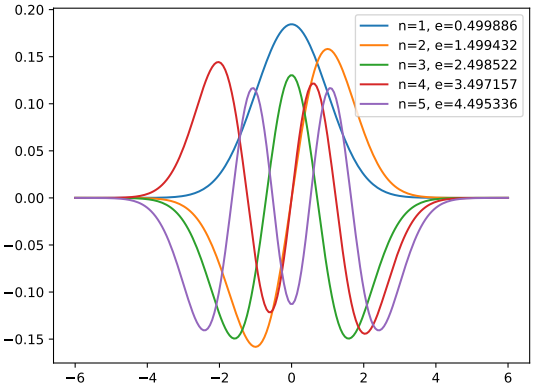

Figure 3: Harmonic oscillator, whose exact energies are $1/2, 3/2, 5/2, 7/2, \ldots$. The calculated values are slightly off due to the finite precision of the grid.

```
# (The number of electrons is twice the number of
#  states -- each state is double occupied.)

x_g = np.linspace(-xmax, xmax, Ng)
dx = x_g[1] - x_g[0]

vext_g = 0.5 * x_g**2  # External potential

T_gg = np.zeros((Ng, Ng))  # Kinetic operator
for i in range(Ng):
    T_gg[i, i] = -2.0
    if i > 0:
        T_gg[i, i - 1] = 1.0
        T_gg[i - 1, i] = 1.0
T_gg *= -0.5 / dx**2

# Initialize density as even:
n_g = 2.0 * Nn / (Ng * dx) * np.ones(Ng)
print('Initial charge', n_g.sum() * dx)

# Nn states, each one doubly occupied.
# Initialize as constant density:
vhartree_g = np.zeros(Ng)
vx_g = np.zeros(Ng)


def soft_poisson_solve(n_g):
    vhartree_g = np.zeros(Ng)
    for i in range(Ng):
        for j in range(Ng):
            vhartree_g[i] += n_g[j] / np.sqrt(1.0 + (x_g[i] - x_g[j])**2)
    vhartree_g *= dx
    Ehartree = 0.5 * (vhartree_g * n_g).sum() * dx
    return Ehartree, vhartree_g


def calculate_exchange(n_g):
    vx_g = -(3.0 / np.pi * n_g)**(1.0 / 3.0)
    Ex_prefactor = -3.0 / 4.0 * (3.0 / np.pi)**(1.0 / 3.0)
    Ex = Ex_prefactor * (n_g**(4.0 / 3.0)).sum() * dx
    return Ex, vx_g


density_change = 1.0
while density_change > 1e-6:
    # Calculate Hamiltonian
    veff_g = vext_g + vhartree_g + vx_g
    H_gg = T_gg + np.diag(veff_g)  # Hamiltonian

    # Solve KS equations
```

```
    eps_n, psi_gn = np.linalg.eigh(H_gg)
    print('Energies', ' '.join('{:4f}'.format(eps)
                               for eps in eps_n[:Nn]))

    # Normalize states.  The states are normalized
    # already, but not in our dx metric
    psi_gn /= np.sqrt(dx)

    # Update density
    nold_g = n_g
    n_g = 2.0 * (psi_gn[:, :Nn]**2).sum(axis=1)
    density_change = np.abs(nold_g - n_g).sum() * dx

    charge = n_g.sum() * dx
    print('Number of electrons', charge)
    print('Convergence err', density_change)
    assert abs(charge - 2.0 * Nn) < 1e-14

    # Calculate Hartree potential
    Ehartree, vhartree_g = soft_poisson_solve(n_g)
    print('Electrostatic energy', Ehartree)

    # Calculate exchange potential
    # (we won't bother with correlation!)
    Ex, vx_g = calculate_exchange(n_g)
    print('Exchange energy', Ex)

    Ebs = 2.0 * eps_n[:Nn].sum()  # "Band structure" energy
    Ekin = Ebs - (veff_g * n_g).sum() * dx
    print('Ekin', Ekin)
    Epot = Ehartree + Ex + (vext_g * n_g).sum() * dx
    print('Epot', Epot)
    Etot = Ekin + Epot
    print('Energy', Etot)


for i in range(Nn):
    plt.plot(x_g, psi_gn[:, i],
             label='n={}, e={:3f}'.format(i + 1, eps_n[i]))
    plt.legend(loc='lower right')
plt.show()
```