Lecture 2: Computing the Adjoint Gradient

This lecture introduces the mathematics of the adjoint method and implements a simple adjoint gradient in Tidy3D, which is compared against the numerical gradient found via finite differences.

We start by importing the packages we need, including Tidy3D.

In [1]:
```
import matplotlib.pylab as plt
import numpy as np

import tidy3d as td
import tidy3d.web as web
```

Setup

First we set up our toy problem.

We assume there is a dielectric rectangle illuminated by a point dipole from the left.

We'll also add a monitor to measure the field at a point to the right of the box.

Our design objective will be to maximize the electric intensity at this point.

In [2]:
```
wavelength = 2.0  # um
freq0 = td.C_0 / wavelength
omega0 = 2 * np.pi * freq0
fwidth = freq0 / 10

# size of simulation domain
Lx = 2.5 * wavelength
Ly = 2.5 * wavelength

# size of rectangle
lx = 1.0 * wavelength
ly = 1.0 * wavelength

# grid size
dl = wavelength / 50

# relative permittivity of box
eps_box = 2.0

# x positions of the source and monitor
pos_src = -0.4 * (Lx / 2 + lx / 2)
pos_mnt = +0.6 * (Lx / 2 + lx / 2)
```

In [3]:
```
# the dielectric box
box = td.Structure(
```

```
    geometry=td.Box(size=(lx, ly, td.inf)),
    medium=td.Medium(permittivity=eps_box),
)

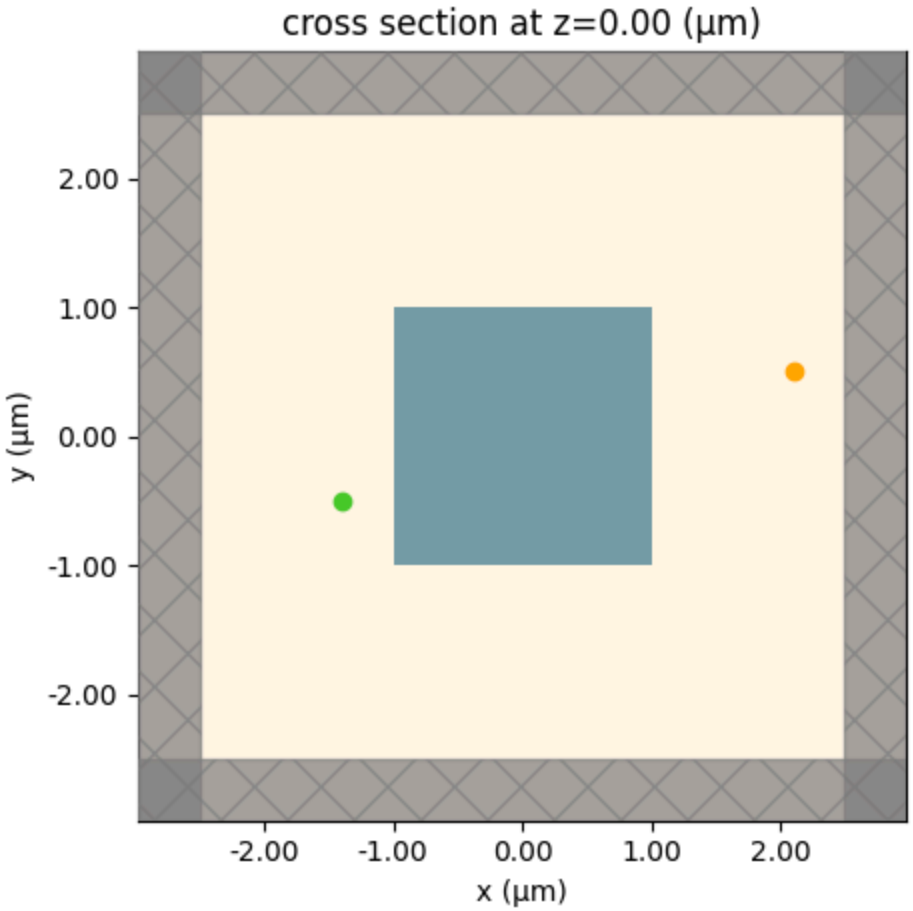
# point dipole source
src = td.PointDipole(
    center=(pos_src, -0.5, 0),
    polarization="Ez",
    source_time=td.GaussianPulse(freq0=freq0, fwidth=fwidth),
)

# monitor to visualize fields in the entire domain
mnt = td.FieldMonitor(size=(td.inf, td.inf, 0), freqs=[freq0], name="field", fields

# monitor to specifically measure the fields at the location of the objective
mnt_objective = td.FieldMonitor(
    size=(0, 0, 0),
    center=(pos_mnt, 0.5, 0),
    freqs=[freq0],
    fields=("Ez",),
    name="measure",
)

# put everything together into a simulation
sim = td.Simulation(
    size=(Lx, Ly, 0),
    structures=[box],
    sources=[src],
    monitors=[mnt, mnt_objective],
    boundary_spec=td.BoundarySpec.pml(x=True, y=True, z=False),
    run_time=100.0 / fwidth,
    grid_spec=td.GridSpec.uniform(dl=dl),
)
```

In [4]:
```
# plot the domain
sim.plot(z=0, monitor_alpha=0.1)
plt.show()
```



In [5]:
```
# run the simulation
sim_data = web.run(sim, task_name="orig", path="data/sim_data_orig.hdf5")
```

```
11:10:12 CEST Created task 'orig' with task_id
              'fdve-27418f7f-e851-4d50-b4b7-5ad409720969' and task_type 'FDTD'.

              View task using web UI at
              'https://tidy3d.simulation.cloud/workbench?taskId=fdve-27418f7f-e8
              51-4d50-b4b7-5ad409720969'.

              Task folder: 'default'.

Output()

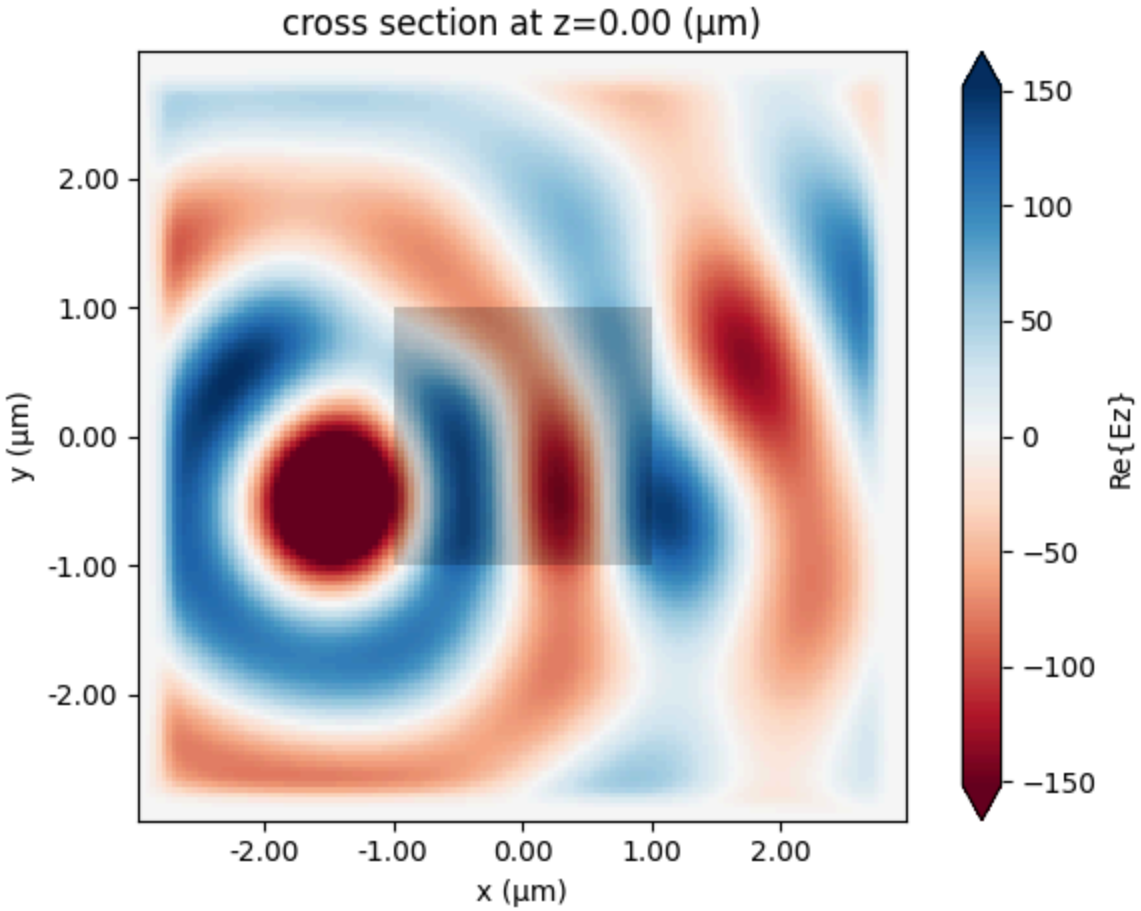
11:10:15 CEST Maximum FlexCredit cost: 0.025. Minimum cost depends on task
              execution details. Use 'web.real_cost(task_id)' to get the billed
              FlexCredit cost after a simulation run.

              status = success

Output()

11:10:17 CEST loading simulation from data/sim_data_orig.hdf5
```

In [6]:
```
# visualize the fields
sim_data.plot_field(field_monitor_name="field", field_name="Ez", val="real")
plt.show()
```



Objective function

Next, we will introduce our objective function, which computes the field intensity at the point to the right of the box.

For convenience, we'll package this as two functions that accept the simulation data.

This makes it easier when we want to re-compute things later.

In [7]:
```
def measure_amplitude(sim_data: td.SimulationData) -> complex:
    """How to post process a SimulationData object to give the intensity at the mea
    field_data = sim_data["measure"]
    ez = field_data.Ez
    return complex(ez)


def measure_intensity(sim_data: td.SimulationData) -> float:
    """How to post process a SimulationData object to give our objective function v
    amplitude = measure_amplitude(sim_data)
    return abs(amplitude) ** 2
```

In [8]:
```
print(f"intensity = {measure_intensity(sim_data):.5f}")
```

```
intensity = 17001.18068
```

Adjoint simulation

As mentioned, our adjoint simulation uses a source that is given by the partial derivative of our objective function with respect to the electric fields.

Our objective function involves measuring E_z at the `mnt_pos` and then squaring the absolute value.

If we write the measurement position as a vector m and E_z as e, then our objective can be written as

$$J(e) = |m^T e|^2 = (m^T e)(m^T e)^*$$

Taking the derivative of J with respect to a design parameter p and respecting the complex-valued nature of e by taking the regular and complex conjugate parts separately, we get:

$$\frac{dJ}{dp} = \frac{\partial J}{\partial e}\frac{de}{dp} + \frac{\partial J}{\partial e^*}\frac{de^*}{dp} = 2\mathcal{R}\{\frac{\partial J}{\partial e}\frac{de}{dp}\} = 2\mathcal{R}\{m^T(m^T e)^*\frac{de}{dp}\}$$

We identify that our adjoint source $\frac{\partial J}{\partial e}$ is given by $m^T(m^T e)^*$.

Let's set up the adjoint simulation now, using our `SimulationData` from the forward pass to compute the adjoint source.

In [9]:
```
# get the "e" from our math before
forward_amp = measure_amplitude(sim_data)

# calculate the complex amplitude of the adjoint source
adj_amp_cplx = -1j * np.conj(forward_amp)

adj_amp = np.abs(adj_amp_cplx)
adj_phase = np.angle(adj_amp_cplx)

# source is located where the measurement monitor was, we apply conj(m^T) to the so
# NOTE: the 1 / MU_0 * omega0 amplitude prefactor is used for normalization and is
src_adj = td.PointDipole(
    center=mnt_objective.center,
    polarization="Ez",
    source_time=td.GaussianPulse(
        freq0=freq0,
        fwidth=fwidth,
        amplitude=adj_amp / (td.MU_0 * omega0),
        phase=adj_phase,
    ),
)

# adjoint simulation is the same as original, but only adjoint source, and remove t
sim_adj = sim.updated_copy(sources=[src_adj], monitors=[mnt])
```

In [10]:
```
sim_data_adj = web.run(sim_adj, task_name="sim_adj", path="data/sim_data_adj.hdf5")
```

```
11:10:18 CEST Created task 'sim_adj' with task_id
              'fdve-3276889b-c626-4cc1-9fda-9a419c68d0af' and task_type 'FDTD'.

              View task using web UI at
              'https://tidy3d.simulation.cloud/workbench?taskId=fdve-3276889b-c6
              26-4cc1-9fda-9a419c68d0af'.

              Task folder: 'default'.

Output()

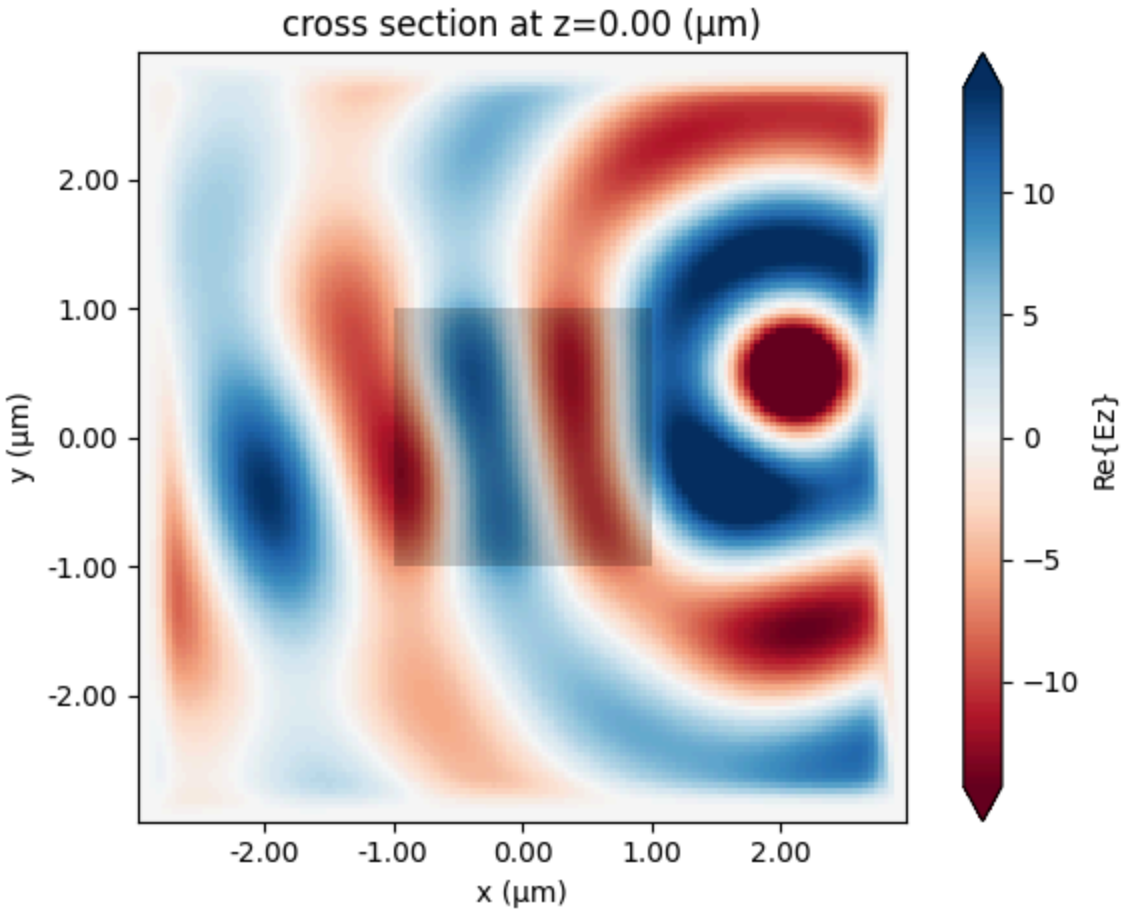
11:10:20 CEST Maximum FlexCredit cost: 0.025. Minimum cost depends on task
              execution details. Use 'web.real_cost(task_id)' to get the billed
              FlexCredit cost after a simulation run.

11:10:21 CEST status = success

Output()

11:10:22 CEST loading simulation from data/sim_data_adj.hdf5
```

In [11]:
```
sim_data_adj.plot_field(field_monitor_name="field", field_name="Ez", val="real")
plt.show()
```



Computing the gradient

Now that we have our forward and adjoint fields, we can multiply them together to get a map of how the change in permittivity at each point will affect our objective function.

In the adjoint-variable method the gradient of a scalar objective J with respect to any design parameter p reduces to:

$$\frac{dJ}{dp} = -2\Re\left\{\mathbf{E}_{\text{fwd}}\,\frac{dA}{dp}\,\mathbf{E}_{\text{adj}}\right\}$$

Here A is the Maxwell operator $A = \nabla \times \nabla \times - \omega^2 \mu_0 \epsilon_r$. The prefactor $2\Re$ accounts for the complex-conjugate symmetry so that the result is a real-valued sensitivity ready for use in gradient-based optimization.

For perturbations that locally change only the relative permittivity ϵ_r, the derivative of the operator with respect to p is simply

$$\frac{dA}{dp} = -\omega^2 \mu_0 \frac{d\epsilon_r}{dp}.$$

If we treat p as the value of ϵ_r in a voxel ($d\epsilon_r/dp = 1$), the gradient map that tells us how J changes when we tweak the permittivity at every point is therefore implemented in code as:

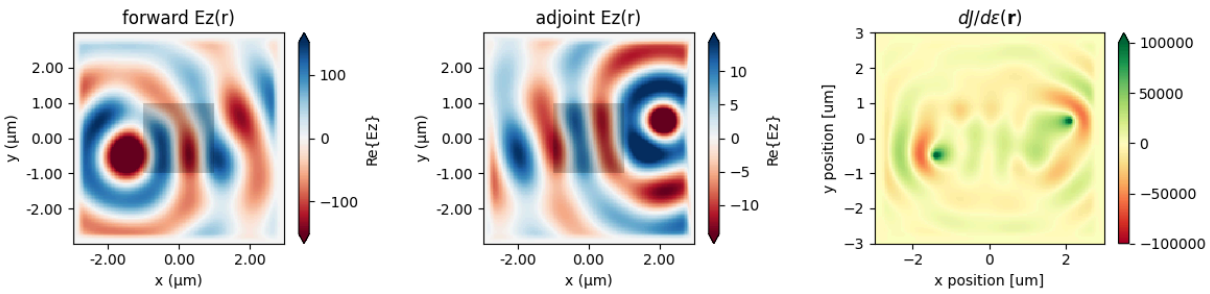
In [12]:
```
e_fwd = sim_data["field"].Ez
e_adj = sim_data_adj["field"].Ez

dA_deps = -(omega0**2) * td.MU_0 * td.EPSILON_0
dJ_deps = -2 * np.real(e_fwd * dA_deps * e_adj)
```

In [13]:
```
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 3), tight_layout=True)
sim_data.plot_field(field_monitor_name="field", field_name="Ez", val="real", ax=ax1
sim_data_adj.plot_field(field_monitor_name="field", field_name="Ez", val="real", ax
dJ_deps.squeeze().plot.pcolormesh(x="x", y="y", cmap="RdYlGn", vmin=-1e5, vmax=1e5,

ax1.set_title("forward Ez(r)")
ax2.set_title("adjoint Ez(r)")

ax3.set_title(r"$dJ/d\epsilon(\mathbf{r})$")
plt.show()
```



Comparison against numerical derivative

To test our implementation, let's imagine our box was made up of four quadrants, each with their own individual permittivities ϵ_i.

We will see how the adjoint gradient compares to one computed with finite difference.

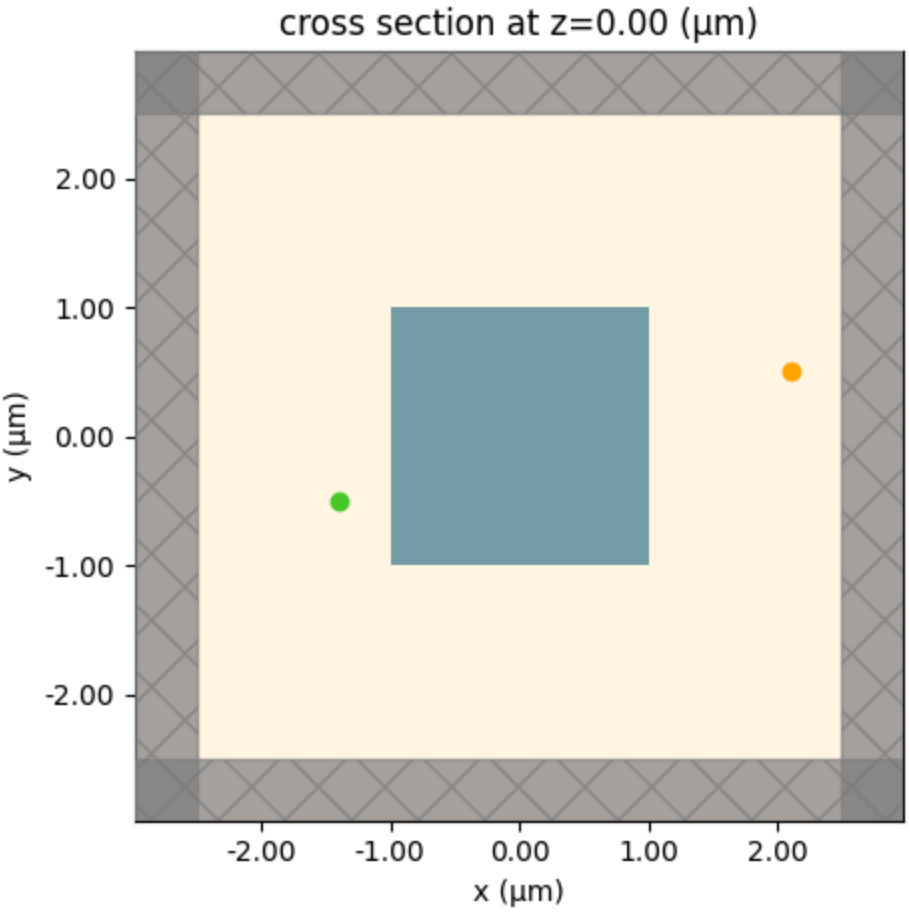
Let's first set up our boxes.

In [14]:
```
boxes = []

for x_center in (-lx / 4, +lx / 4):
    for y_center in (-ly / 4, +ly / 4):
        box = td.Structure(
            geometry=td.Box(center=(x_center, y_center, 0), size=(lx / 2, ly / 2, t
            medium=td.Medium(permittivity=eps_box),
        )
        boxes.append(box)

sim_num = sim.updated_copy(structures=boxes)

ax = sim_num.plot(z=0, monitor_alpha=0.1)
```



Numerical gradient

As a reminder, the numerical derivative with respect to parameter `p` is approximated by

$$\frac{dJ}{dp} \approx \frac{J(p+\Delta) - J(p-\Delta)}{2\Delta}$$

Here, we will compute this for each p in our design variables.

We must loop over each of our box quadrants, perturb the permittivity in that quadrant by a small amount in both plus and minus directions, compute the objective function through a simulation, and take the difference.

This will require 2 N = 8 simulations total, which we'll perform in a batch since they don't depend on each other.

In [15]:
```
# step size
delta = 5e-3

# set up all simulations in the batch with perturbed permittivities
num_grad_sims = {}
for i_box in range(len(boxes)):
    for pm_str, delta_eps in zip("+-", (+delta, -delta)):
        task_name = f"box_{i_box + 1}_{pm_str}"
        new_boxes = list(boxes)
        new_medium = boxes[i_box].medium
        new_medium = new_medium.updated_copy(permittivity=new_medium.permittivity +
        new_boxes[i_box] = boxes[i_box].updated_copy(medium=new_medium)

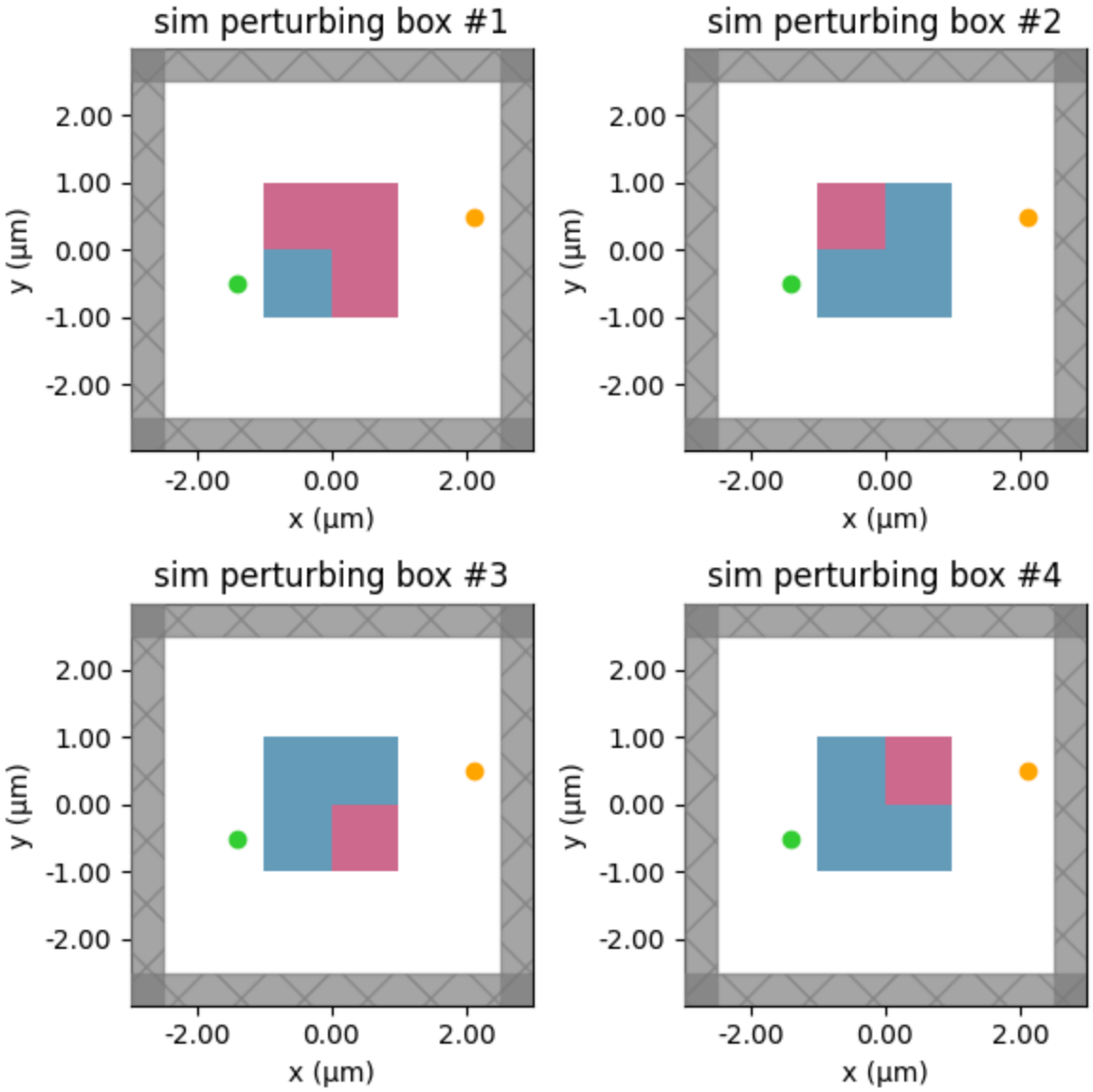
        print(
            f'task_name = "{task_name}": box epsilons = {[b.medium.permittivity for
        )

        new_sim_i = sim.updated_copy(structures=new_boxes)

        num_grad_sims[task_name] = new_sim_i
```

```
task_name = "box_1_+": box epsilons = [2.005, 2.0, 2.0, 2.0]
task_name = "box_1_-": box epsilons = [1.995, 2.0, 2.0, 2.0]
task_name = "box_2_+": box epsilons = [2.0, 2.005, 2.0, 2.0]
task_name = "box_2_-": box epsilons = [2.0, 1.995, 2.0, 2.0]
task_name = "box_3_+": box epsilons = [2.0, 2.0, 2.005, 2.0]
task_name = "box_3_-": box epsilons = [2.0, 2.0, 1.995, 2.0]
task_name = "box_4_+": box epsilons = [2.0, 2.0, 2.0, 2.005]
task_name = "box_4_-": box epsilons = [2.0, 2.0, 2.0, 1.995]
```

In [16]:
```
# inspect each of the forward perturbed sims to get a feeling for what is being sim
_, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, tight_layout=True, figsize=(6, 6))
axes = [ax1, ax2, ax3, ax4]
for ax, box_num in zip(axes, "1234"):
    num_grad_sims[f"box_{box_num}_+"].plot(z=0, monitor_alpha=0.0, ax=ax)
    ax.set_title(f"sim perturbing box #{box_num}")
```



In [17]:
```
# run all simulations in a batch
batch_data = web.run_async(simulations=num_grad_sims, path_dir="data")
```

```
Output()
```

```
11:10:26 CEST Started working on Batch containing 8 tasks.
11:10:33 CEST Maximum FlexCredit cost: 0.200 for the whole batch.
              Use 'Batch.real_cost()' to get the billed FlexCredit cost after
              the Batch has completed.
```

```
Output()
11:10:39 CEST Batch complete.
```

```
Output()
```

In [18]:
```
# post process batch results using finite difference derivative equation to compute
grad_num = np.zeros(len(boxes))
for task_name, sim_data in batch_data.items():
    _, box_i_plus1, pm_str = task_name.split("_")
```

```
    box_i = int(box_i_plus1) - 1
    constant = 1 / (2 * delta) if pm_str == "+" else -1 / (2 * delta)
    J_i = measure_intensity(sim_data)
    grad_num[box_i] += constant * J_i
```

Adjoint gradient

For the adjoint gradient, we simply sum our permittivity derivative map over each of the domains associated with the permittivities.

Note, the neat thing about this is that we **only require 2 simulations** to compute the gradient, no matter how many design parameters we have.

The permittivity derivative matp contains all the information we need to compute the gradient w.r.t. a set of arbitrary parameters.

In [19]:
```
grad_adj = []

num_pts = 100

# loop over quadrants and integrate using trapezoidal rule
for x_center in (-lx / 4, +lx / 4):
    dx = lx / 2 / num_pts
    x_bounds = (x_center - lx / 4, x_center + lx / 4)
    xpts = np.linspace(x_bounds[0] + dx / 2, x_bounds[1] - dx / 2, num_pts)
    for y_center in (-ly / 4, +ly / 4):
        dy = ly / 2 / num_pts
        y_bounds = (y_center - ly / 4, y_center + ly / 4)
        ypts = np.linspace(y_bounds[0] + dy / 2, y_bounds[1] - dy / 2, num_pts)
        grad_adj_i = np.sum(dJ_deps.interp(x=xpts, y=ypts)) * dx * dy
        grad_adj.append(grad_adj_i)

grad_adj = np.array(grad_adj)
```

In [20]:
```
print("adjoint gradient = ", grad_adj)
print("numerical gradient = ", grad_num)
```

```
adjoint gradient =  [ 6899.22349859 -3010.03007895  7095.60595275  -651.46163376]
numerical gradient =  [ 6882.71479218 -3020.51370814  7088.20877809  -671.17994514]
```

We compare the two gradients and see that they match quite well! to 0.27% RMS error.

In [21]:
```
rms_unnormalized = np.linalg.norm(grad_adj - grad_num) / np.linalg.norm(grad_num)
print(f"rms unnormalized = {(rms_unnormalized * 100):.4f} %")
```

```
rms unnormalized = 0.2776 %
```

For the purposes of optimization, we often care more about the **direction** of the gradient vector than it's overall magnitude.

This is because most optimization algorithms require the user to specify a step size for iteration anyway.

In this spirit, comparing the adjoint and numerical gradient vectors normalized as unit vectors, the error gets even smaller.

In [22]:
```
rms_normalized = np.linalg.norm(
    grad_adj / np.linalg.norm(grad_adj) - grad_num / np.linalg.norm(grad_num)
)
print(f"rms normalized = {(rms_normalized * 100):.4f} %")
```

```
rms normalized = 0.2533 %
```

This is a typical approach towards checking that your adjoint code is working properly.

In the following notebooks, we will make use of the native automatic differentiation support in tidy3d to compute these gradients without all of this extra work, and use it to do some interesting optimizations.