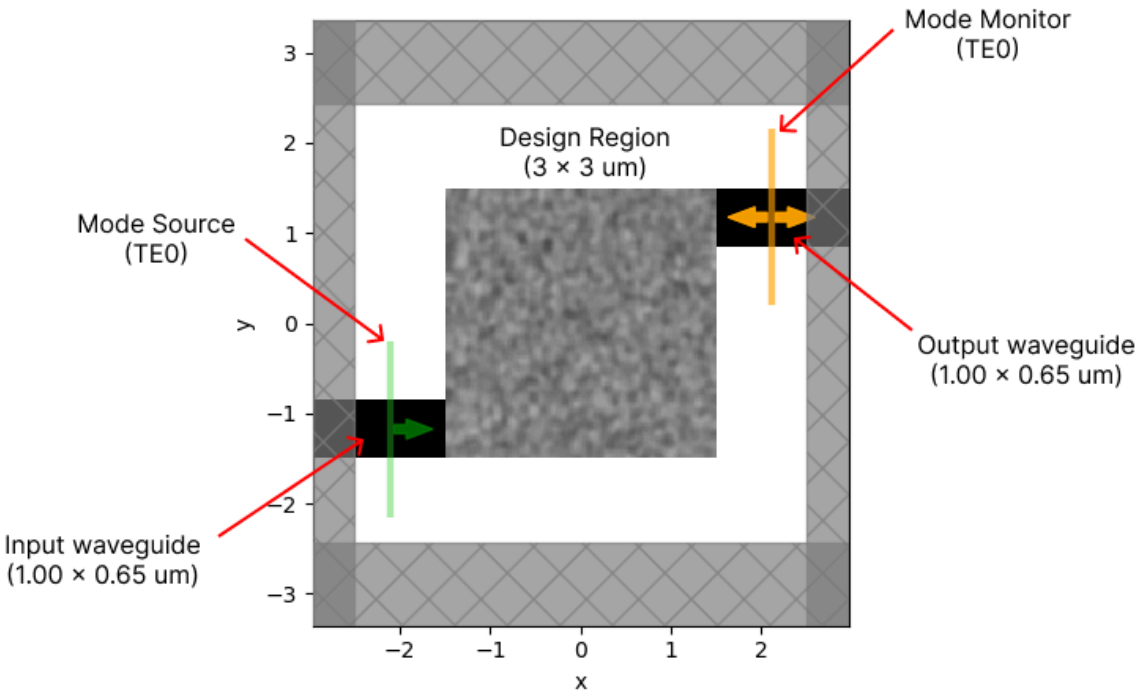
Session 5 Challenge - Inverse Design in Photonics

"Inverse design" is a technique used for the automated design of photonic devices. The process involves mathematically defining a figure of merit for one's device and then using gradient-based optimization to maximize this function efficiently for a large number of design parameters. Using the "adjoint method," you can compute gradients of a figure-of-merit with respect to the design parameters using only two simulations (forward and adjoint), no matter how many design parameters are present. This gradient is then used for a high-dimensional, gradient-based system optimization.

Challenge Description

In this challenge, we propose you build a topology optimization-based adjoint inverse design to create a waveguide s-bend. The image below shows the optimization setup you will build, including the input and output waveguides, the design region, the source, and monitor position. To reduce the total optimization time, you will run 2D FDTD simulations, however, you can easily adapt this exercise to build 3D optimization models.



Follow the next steps to build the adjoint optimization setup and run the inverse design optimization.

Notes

> You can develop this example using the Tidy3D Python notebook.

> To install the `jax` module required for this challenge, we recommend running `pip install "tidy3d[jax]"` .

Step 1: Import Python Packages

In [ ]:

```
# Standard python imports.
from typing import List
import numpy as np
import scipy as sp
import matplotlib.pylab as plt
import optax

# Import jax to be able to use automatic differentiation.
import jax.numpy as jnp
from jax import value_and_grad

# Import regular tidy3d.
import tidy3d as td
import tidy3d.web as web
import tidy3d.plugins.adjoint as tda
from tidy3d.plugins.mode import ModeSolver
from tidy3d.plugins.mode.web import run as run_mode_solver
```

Step 2: Parameter Definition

Below, you have the definition of all the parameters you will need throughout the optimization. The design region is a 3 μm x 3 μm rectangle. Within the design region, the refractive index is allowed to vary between 1.0 and 2.0. The input and output waveguides are 1 μm long, 0.65 μm wide, and have a refractive index of 2.

You do not need to change the parameters below!

In [ ]:

```
# Geometric parameters.
w_width = 0.65  # Input/Output waveguide width (um).
w_length = 1.0  # Input/Output waveguide length (um).
w_thick = td.inf  # Structure dimension in z-direction (infinity in 2D simula

# Material.
n = 2.00  # Material refractive index.

# Design region parameters.
dr_width = 3.0  # Design region width (um).
dr_length = 3.0  # Design region length (um).
dr_grid_size = 0.02  # Grid size within the design region (um).

# Inverse design set up parameters.
eta = 0.50  # Threshold value for the projection filter.
min_feature_size = 0.080
beta_min = 1  # Minimum projection slope parameter.
beta_step = 1  # Increments to the projection slope parameter throughout the
num_steps = 20  # Number of optimization steps.
learning_rate = 1.0  # Optimizer hyperparameter.

# Simulation wavelength.
wl = 1.55  # Central simulation wavelength (um).
bw = 0.10  # Simulation bandwidth (um).
n_wl = 101  # Number of wavelength points within the bandwidth (to monitor se
```

Next, calculated parameters are defined.

In [ ]:

```
# Minimum and maximum values for the permittivities.
eps_max = n ** 2
eps_min = 1.0

# Material definitions.
mat = td.Medium(permittivity=eps_max)  # Waveguide material.

# Wavelengths and frequencies.
wl_max = wl + bw / 2
wl_min = wl - bw / 2
wl_range = np.linspace(wl_min, wl_max, n_wl)
freq = td.C_0 / wl
freqs = td.C_0 / wl_range
freqw = 0.5 * (freqs[0] - freqs[-1])
run_time = 2e-12

# Computational domain size.
pml_spacing = 0.6 * wl
size_x = 2 * w_length + dr_length
size_y = dr_width + 2 * pml_spacing
size_z = 0
eff_inf = 10

# Inverse design variables.
nx = int(dr_length / dr_grid_size)
ny = int(dr_width / dr_grid_size)
npar = int(nx * ny)
dr_size_x = nx * dr_grid_size
dr_size_y = ny * dr_grid_size

# Initial design region parameters.
init_par = np.random.uniform(0, 1, int(npar))
init_par = sp.ndimage.gaussian_filter(init_par, 1)
init_par = init_par.reshape((nx, ny))
```

Step 3: Static Geometry

Define the static structures. In this case, they are the input and output s-bend waveguides. So, create the `input_waveguide` and `output_waveguide` structure objects using the `from_bounds` method. Remember to extend the waveguides beyond the simulation domain to make the invariant within the PML layers.

In [ ]:

Step 4: Mode Analysis

(a) Create a `Simulation` object including only the input waveguide, define a source plane perpendicular to the waveguide propagation direction, and run a `ModeSolver` to calculate the first 4 waveguide modes.

In [ ]:

(b) Plot the mode field components `Ex` , `Ey` , and `Ez` for all the calculated modes. Then, identify the fundamental TE waveguide mode.

In [ ]:

Step 5: Create the Input Source and the Output Monitor

(a) Create a `ModeSource` object to inject the fundamental TE waveguide mode at the input waveguide. Use the `freq` and `freqw` parameters to set a `GaussianPulse` and assign it to `source_time` . In addition to adequately setting `mode_index` to the fundamental TE value, the `num_modes` parameter of `ModeSpec` must be calculated as `mode_index + 1` .

In [ ]:

(b) Define a `ModeMonitor` at the output waveguide. You will use this monitor to calculate the optimization figure of merit. You can use the same `ModeSpec` object previously defined for the source. Set the monitor's `freqs` parameter to the central simulation frequency `freq` , and the `name` parameter to "fom". Additionally, create a second `ModeMonitor` similar to the latter one, but set its `freqs` parameter to the whole simulation frequency range (`freqs`). You will obtain the final s-bend spectral response using this monitor.

In [ ]:

(c) Create a `FieldMonitor` to record the electromagnetic fields at the `xy` plane. Set the monitor parameter `freqs` to the central simulation frequency (`freq`).

In [ ]:

Step 6: Update Design Region

Next, we have defined filter and projection functions to include fabrication constraints over the optimization process.

In [ ]:
```
from tidy3d.plugins.adjoint.utils.filter import ConicFilter, BinaryProjector

conic_filter = ConicFilter(radius=min_feature_size, design_region_dl=dr_grid_

def filter_project(params, beta, eta=0.5):
    """Apply conic filter and binarization to the raw params."""
    params_smooth = conic_filter.evaluate(params)
    binary_projector = BinaryProjector(vmin=0, vmax=1, beta=beta, eta=eta)
    params_smooth_binarized = binary_projector.evaluate(params_smooth)
    return params_smooth_binarized

def get_eps(params, beta):
    """Get the permittivity values (eps_min, eps_max) array as a funciton of
    processed_params = filter_project(params, beta)
    return eps_min + processed_params * (eps_max - eps_min)
```

Here, you must implement the `update_design_region` function to create a `JaxCustomMedium` based on the design parameters (`params`) and then return a `JaxStructure` .

In [ ]:
```
def update_design_region(params, beta) -> List[tda.JaxStructure]:
    """Create the design region based on the design parameters."""

    # Your code goes here!!
    # design_structure must be a tda.JaxStructure object!

    return [design_structure]
```

Step 7: Jax Simulation

(a) Now, implement the `build_adjoint_sim` function to return a `JaxSimulation` object. This function will receive the design parameters (`params`) and the projection slope parameter (`beta`). You need to call the `update_design_parameter` function and arrange all the simulation objects defined before into the `JaxSimulation` object.

In [ ]:
```
def build_adjoint_sim(params, beta) -> tda.JaxSimulation:
    """Create the adjoint simulation based on the design parameters."""

    # Your code goes here!!

    return tda.JaxSimulation(
        size=[size_x, size_y, size_z],
        grid_spec=,  # Fill in the grid_spec.
        structures=[],  # Fill in the structures.
        input_structures=,  # Fill in the input_structures.
        sources=[],   # Fill in the sources.
        monitors=[],   # Fill in the monitors.
```

```
        output_monitors=[],   # Fill in the output_monitors.
        run_time=run_time,
        boundary_spec=td.BoundarySpec.pml(x=True, y=True, z=False),
    )
```

(b) Call `build_adjoint_sim` to create an initial simulation object and plot it to verify the setup.

In [ ]:

Step 8: Objective Function

The optimization setup is almost ready!

(a) Implement the objective function (`objective_function`) used within the optimization process. This function must receive the design parameters and the projection slope parameter and return a single `float` value. So, here you need to create and run the simulation, get the mode amplitude from the `fom` monitor, and calculate and return the modal power.

In [ ]:
```
def objective_function(params, beta: float, step_num: int = None, verbose: bo
    """Create and run the adjoint simulation, then calculate and return the c

    # Your code goes here!!

    return jnp.sum(jnp.abs(amp) ** 2)
```

(b) Use the `Jax.value_and_grad` function to get the gradient of `objective_function` with respect to the permittivity while also returning the coupling efficiency associated with the current iteration so you can record this value for later. If everything is correct in your setup, you will successfully run the forward and adjoint optimizations. Then, print the objective function value and gradients.

In [ ]:
```
df_dn = value_and_grad(objective_function)
coup_eff, grad = df_dn(init_par, beta=beta_min, verbose=False)
print(f"Coupling Efficiency: {coup_eff:.3f}")
print("Gradient values:")
print(grad)
```

Step 9: Run the Optimization

After successfully executing the previous steps, run the code below to start the optimization.

In [ ]:
```
# Initialize Adam optimizer with starting parameters.
params = np.array(init_par)
optimizer = optax.adam(learning_rate=learning_rate)
opt_state = optimizer.init(params)

# Store history.
obj_history = []
params_history = [params]
beta_history = []
```

```
for i in range(num_steps):

    # Compute the gradient and the objective function value.
    beta = beta_min + i * beta_step
    value, gradient = df_dn(params, step_num=i+1, beta=beta)

    # Print the values to the output.
    print(f"Iteration = {i + 1}")
    print(f"\tCoupling Efficiency = {value:.4e}")
    print(f"\tGrad_norm = {np.linalg.norm(gradient):.4e}")
    print(f"\tBeta = {beta:.4e}")

    # Compute and apply updates to the optimizer based on gradient (-1 sign t
    updates, opt_state = optimizer.update(-gradient, opt_state, params)
    params = optax.apply_updates(params, updates)

    # Cap the parameters.
    params = jnp.minimum(params, 1.0)
    params = jnp.maximum(params, 0.0)

    # save history
    obj_history.append(value)
    params_history.append(params)
    beta_history.append(beta)
```

Step 10: Results

In this last step, build and run a simulation using the optimization results. Then, plot the following charts:

- Coupling Efficiency x Iterations.
- Final structure (use `plot_eps`).
- Coupling Efficiency x Wavelength.
- Electric field intensity distribution.

> To build the the final simulation you can use `sim_final = build_adjoint_sim(params_history[-1], beta=beta_history[-1])` using the final optimization parameters and so, use `sim_final.to_simulation()[0]`.

In [ ]:

Run the final simulation.

In [ ]:

Plot the results and upload the chart to earn the FlexCredits!

In [ ]:

Optional Step: Export to GDS

As an optional step, you can also export the inverse design structure to GDS!

In [ ]: