Using Spins

While the SPINS-client graphical user interface is a convenient choice for established problems such as 1D grating couplers and waveguide-based devices, sometimes additional design flexibility is required.

Here we provide a closer look at the examples and go over the usage of the SPINS Python API.

Running an optimization in SPINS python API consists of 4 steps:

1. Creating a simulation space
2. Making the objecetive function
3. Setting the transformations
4. Running the optimization

These four steps seen in the code as the following:

```
def main() -> None:
    """Runs the optimization."""

    # Create the simulation space using the GDS files.
    sim_space = create_sim_space("sim_fg.gds", "sim_bg.gds")

    # Setup the objectives and all values that should be recorded (monitors).
    obj, monitors = create_objective(sim_space)

    # Create the list of operations that should be performed during
    # optimization. In this case, we use a series of continuous parametrizations
    # that approximate a discrete structure.
    trans_list = create_transformations(
        obj, monitors, sim_space, cont_iters=100, min_feature=100)

    # Execute the optimization and indicate that the current folder (".") is
    # the project folder. The project folder is the root folder for any
    # auxiliary files (e.g. GDS files). By default, all log files produced
    # during the optimization are also saved here. This can be changed by
    # passing in a third optional argument.
    plan = optplan.OptimizationPlan(transformations=trans_list)
    problem_graph.run_plan(plan, ".")
```

Below, we go over each of these steps in detail.

Simulation Space

As seen in the examples for the grating coupler and wavelength demultiplexer, the first step of optimization creates the **`spins.invdes.problem_graph.simspace`**. The arguments to the function are the `eps_fg.gds` and `eps_bg.gds` files.

Foreground/Background Permittivity

The files `eps_fg.gds` and `eps_bg.gds` are used to construct the "selection matrix." While the selection matrix is used in many contexts in SPINS, the surface-level purpose of the object is to denote the optimization

region.

This region over which optimization is to take place is given as the XOR between the regions specified by the `eps_fg.gds` and `eps_bg.gds`.

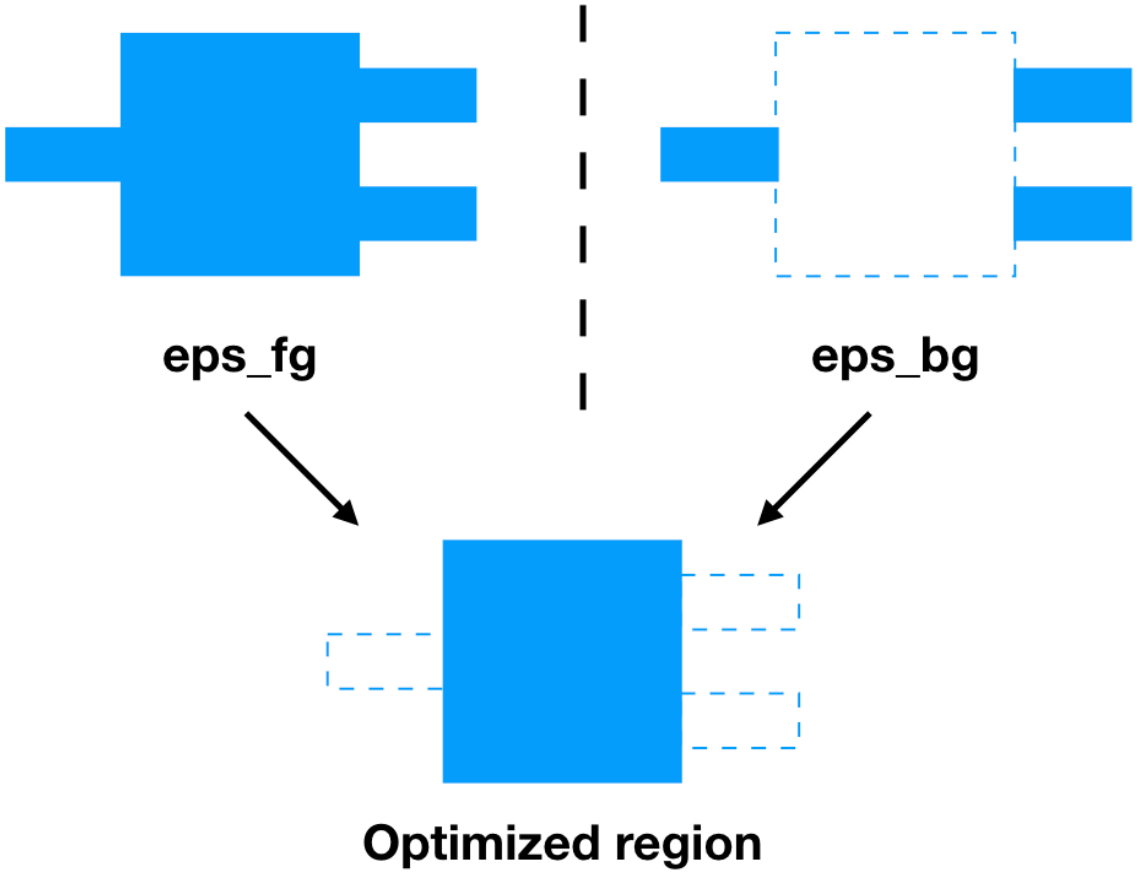

Diagram showing how the optimized regions are specified using the foreground and background permittivities.

The actual operations done to construct the selection matrix from these two GDS files is abstracted away. However, to complete the description of the **`spins.invdes.problem_graph.simspace`**, additional information on the material stack is required.

Material Stack and Design Layer

To be able to construct the entire permittivity distribution from the GDS files, **`spins.invdes.problem_graph.optplan.schema_em.GdsEps`**, we must specify the **`spins.invdes.problem_graph.optplan.schema_em.GdsMaterialStack`**.

A single material is specified through

```
mat_oxide = optplan.Material(index=optplan.ComplexNumber(real=1.5))
```

where **spins.invdes.problem_graph.optplan.optplan.ComplexNumber** provides the utility to specify the `real` and `imag` parts of a material.

The material stack is specified below

```
mat_stack = optplan.GdsMaterialStack(
    background=mat_oxide,
    stack=[
        optplan.GdsMaterialStackLayer(
            foreground=mat_oxide,
            background=mat_oxide,
            gds_layer=[101, 0],
            extents=[-10000, -110],
        ),
        optplan.GdsMaterialStackLayer(
            foreground=optplan.Material(
                index=optplan.ComplexNumber(real=device_index)),
            background=mat_oxide,
            gds_layer=[100, 0],
            extents=[-110, 110],
        ),
    ],
)
```

where the stack is specified a list of **spins.invdes.problem_graph.optplan.schema_em.GdsMaterialStackLayer** objects. The `foreground` material specifies the material of all features explicitly defined in this layer (through a GDS file, for example) and the `background` index is for the remaining regions. The `gds_layer=[100, 0]` (GDS layer 100, datatype 0) specifies the design layer, the layer being optimized over, which is specially marked with layer `[100, 0]`. The value for `gds_layer` for the other material layers of the stack are arbitrary and the user can choose any value between `101-199` for the GDS layer (the first argument), while the GDS datatype (the second argument), should be kept `0`.

Lastly, the `extent` specifies the 3D layer height, and is specified in nanometers.

To completely specify the **spins.invdes.problem_graph.simspace**, all that remains is denoting the simulation region, as done below

```
if SIM_2D:
    # If the simulation is 2D, then we just take a slice through the
    # device layer at z = 0. We apply periodic boundary conditions along
    # the z-axis by setting PML thicknes to zero.
    sim_region = optplan.Box3d(
        center=[0, 0, 0], extents=[5000, 5000, GRID_SPACING])
    pml_thickness = [10, 10, 10, 10, 0, 0]
else:
    sim_region = optplan.Box3d(center=[0, 0, 0], extents=[5000, 5000, 2000])
    pml_thickness = [10, 10, 10, 10, 10, 10]

return optplan.SimulationSpace(
    name="simspace_cont",
    mesh=optplan.UniformMesh(dx=GRID_SPACING),
    eps_fg=optplan.GdsEps(gds=gds_fg, mat_stack=mat_stack),
    eps_bg=optplan.GdsEps(gds=gds_bg, mat_stack=mat_stack),
    sim_region=sim_region,
```

```
        selection_matrix_type="direct_lattice",
        boundary_conditions=[optplan.BlochBoundary()] * 6,
        pml_thickness=pml_thickness,
    )
```

Objective Functions

Overlaps

SPINS provides objects such as **`spins.invdes.problem_graph.optplan.schema_em.Overlap`** which inherently support operations such as addition, subtraction, scalar multiplication, and absolute value, **`spins.invdes.problem_graph.optplan.schema_function.abs`**. This functionality assists in the creation of objective functions which often take the form of overlap integrals between simulated fields and a target mode.

This usage of operations on the **`spins.invdes.problem_graph.optplan.schema_em.WaveguideModeOverlap`** and **`spins.invdes.problem_graph.optplan.schema_em.Overlap`** object to build an objective function can be seen here

```
    # Create modal overlaps at the two output waveguides.
    overlap_1550 = optplan.WaveguideModeOverlap(
        center=[1730, -500, 0],
        extents=[GRID_SPACING, 1500, 600],
        mode_num=0,
        normal=[1, 0, 0],
        power=1.0,
    )
    overlap_1300 = optplan.WaveguideModeOverlap(
        center=[1730, 500, 0],
        extents=[GRID_SPACING, 1500, 600],
        mode_num=0,
        normal=[1, 0, 0],
        power=1.0,
    )
```

which are used to define the objective function below

```
    for wlen, overlap in zip([1300, 1550], [overlap_1300, overlap_1550]):

        ...

        overlap = optplan.Overlap(simulation=sim, overlap=overlap)

        power = optplan.abs(overlap)**2
        power_objs.append(power)
        monitor_list.append(
            optplan.SimpleMonitor(name="power{}".format(wlen), function=power))

    # Spins minimizes the objective function, so to make `power` maximized,
    # we minimize `1 - power`.
    obj = 0
    for power in power_objs:
```

```
    obj += (1 - power)**2
```

Here, an overlap is created, modified to relate to the power through the waveguide, and then turned into an objective function. Note, while these operations are taking place, the final `obj` object is still of type **`spins.invdes.problem_graph.optplan.schema_em.Overlap`**.

Simulation and Sources

As seen above, the **`spins.invdes.problem_graph.optplan.schema_em.Overlap`** object requires a **`spins.invdes.problem_graph.optplan.schema_em.FdfdSimulation`** to provide information regarding the `source`, `solver`, `wavelength`, `sim_space`, and permittivity `epsilon` to be used.

We define a `sim` object for each **`spins.invdes.problem_graph.optplan.schema_em.WaveguideModeOverlap`** object, and use the `sim` object in the arguments for the **`spins.invdes.problem_graph.optplan.schema_em.Overlap`**.

```
power_objs = []
# Keep track of metrics and fields that we want to monitor.
monitor_list = []
for wlen, overlap in zip([1300, 1550], [overlap_1300, overlap_1550]):
    epsilon = optplan.Epsilon(
        simulation_space=sim_space,
        wavelength=wlen,
    )
    sim = optplan.FdfdSimulation(
        source=wg_source,
        # Use a direct matrix solver (e.g. LU-factorization) on CPU for
        # 2D simulations and the GPU Maxwell solver for 3D.
        solver="local_direct" if SIM_2D else "maxwell_cg",
        wavelength=wlen,
        simulation_space=sim_space,
        epsilon=epsilon,
    )
    # Take a field slice through the z=0 plane to save each iteration.
    monitor_list.append(
        optplan.FieldMonitor(
            name="field{}".format(wlen),
            function=sim,
            normal=[0, 0, 1],
            center=[0, 0, 0],
        ))
    if wlen == 1300:
        # Only save the permittivity at 1300 nm because the permittivity
        # at 1550 nm is the same (as a constant permittivity value was
        # selected in the simulation space creation process).
        monitor_list.append(
            optplan.FieldMonitor(
                name="epsilon",
                function=epsilon,
                normal=[0, 0, 1],
                center=[0, 0, 0]))

    overlap = optplan.Overlap(simulation=sim, overlap=overlap)

    power = optplan.abs(overlap)**2
```

```
        power_objs.append(power)
        monitor_list.append(
            optplan.SimpleMonitor(name="power{}".format(wlen), function=power))
```

Additionally, a **spins.invdes.problem_graph.optplan.schema_em.WaveguideModeSource** is specified at the start of the function.

```
    wg_source = optplan.WaveguideModeSource(
        center=[-1770, 0, 0],
        extents=[GRID_SPACING, 1500, 600],
        normal=[1, 0, 0],
        mode_num=0,
        power=1.0,
    )
```

Other sources can be used as well, for example a gaussian beam source is specified through the **spins.invdes.problem_graph.optplan.schema_em.GaussianSource** class, and is used in the grating coupler example:

```
        source=optplan.GaussianSource(
        polarization_angle=0,
        theta=np.deg2rad(-10),
        psi=np.pi / 2,
        center=[0, 0, wg_thickness + 700],
        extents=[14000, 14000, 0],
        normal=[0, 0, -1],
        power=1,
        w0=5200,
        normalize_by_sim=True,
    ),
```

Monitors

While not necessary to the optimization, **spins.invdes.problem_graph.optplan.schema_monitor.SimpleMonitor** are used to specify which quantities or fields are to be saved for later viewing by the viewer.

spins.invdes.problem_graph.optplan.schema_monitor.FieldMonitor is intended for fields like electric fields and permittivity distributions. The distinction with `FieldMonitor` vs `SimpleMonitor` is that they can take slices and thus save a smaller amount of information. `SimpleMonitor` saves all values recorded by the field, whatever they are and so are typically used for scalars

In the example code above, we see that the fields are saved for both wavelengths, the permittivity for a single wavelength, and the power through the waveguide for both wavelengths.

Transformations

Transformations are what orchestrate the optimization in SPINS. Each **spins.invdes.problem_graph.optplan.optplan.Transformation** is appended to a `trans_list`, transformation list, which is run in series by the code. A transformation can specify how to run a stage of optimization (continuous, discrete) or how to convert the parameterization from one form to another.

In the wavelength demultiplexer example, we see both use cases during continuous optimization. But first, we need to define the `trans_list` and an initial parameterization.

```
# Setup empty transformation list.
trans_list = []

# First do continuous relaxation optimization.
# This is done through cubic interpolation and then applying a sigmoid
# function.
param = optplan.CubicParametrization(
    # Specify the coarseness of the cubic interpolation points in terms
    # of number of Yee cells. Feature size is approximated by having
    # control points on the order of `min_feature / GRID_SPACING`.
    undersample=3.5 * min_feature / GRID_SPACING,
    simulation_space=sim_space,
    init_method=optplan.UniformInitializer(min_val=0.6, max_val=0.9),
)
```

Continuous Optimization

In the wavelength demultiplexer example, only continuous optimization is carried out. However, during the optimization, a sigmoid ramp is applied to the permittivity distribution, in order to bias the continuous values towards more discrete structures.

In the example code this is seen through iterating over `num_stages`, where a stage of optimization is followed by **`spins.invdes.problem_graph.optplan.schema_opt.CubicParamSigmoidStrength`** to update the parameterization.

```
iters = max(cont_iters // num_stages, 1)
for stage in range(num_stages):
    trans_list.append(
        optplan.Transformation(
            name="opt_cont{}".format(stage),
            parametrization=param,
            transformation=optplan.ScipyOptimizerTransformation(
                optimizer="L-BFGS-B",
                objective=obj,
                monitor_lists=optplan.ScipyOptimizerMonitorList(
                    callback_monitors=monitors,
                    start_monitors=monitors,
                    end_monitors=monitors),
                optimization_options=optplan.ScipyOptimizerOptions(
                    maxiter=iters),
            ),
        ))

    if stage < num_stages - 1:
        # Make the structure more discrete.
        trans_list.append(
            optplan.Transformation(
                name="sigmoid_change{}".format(stage),
                parametrization=param,
                # The larger the sigmoid strength value, the more "discrete"
                # structure will be.
                transformation=optplan.CubicParamSigmoidStrength(
```

```
                    value=4 * (stage + 1)),
            ))
    return trans_list
```

Discrete Optimization

While not provided in open-source SPINS-B, the method to optimize with a discrete parameterization (level-set parameterization) is also done by appending elements into the `trans_list`. First a transformation to convert from the continuous parameterization to level-set parameterization is appended, followed by optimization iterations with the updated parameterization.

Running Optimization

With a simulation space, objective function, and transformation list defined, we can begin our optimization!

At this point we can generate the computational graph completely describing the optimization with the line:

```
# Execute the optimization and indicate that the current folder (".") is
# the project folder. The project folder is the root folder for any
# auxiliary files (e.g. GDS files). By default, all log files produced
# during the optimization are also saved here. This can be changed by
# passing in a third optional argument.
plan = optplan.OptimizationPlan(transformations=trans_list)
```

and subsequently run the graph with

```
problem_graph.run_plan(plan, ".")
```

This will begin optimization as specified by the `trans_list`, saving as the output the monitors defined earlier. If the parameterization supports it, additionally a GDS file of the final permittivity will be saved as well.

Examples

Wavelength Demultiplexer

The example code and descriptions found above are used in the context of the following files:

wdm.py

sim_fg.gds

sim_bg.gds

```
1     """Optimizes a 2-way demultiplexer.
2
```

```
    This example shows how to optimize 2 um x 2 um 2-way demultiplexer that splits
    1550 nm and 1300 nm light. This is shown diagrmatically below:


            _______
           |       |___
       ---         ____ out0
    in ---         |___
           |       ____ out1
           |_______|

    By changing the `SIM_2D` global variable, the simulation can be done in either
    2D or 3D. 2D simulations are performed on the CPU whereas 3D simulations require
    using the GPU-based Maxwell electromagnetic solver.

    Note that to run the 3D optimization, the 3D solver must be setup and running
    already.

    To process the optimization data, see the IPython notebook contained in this
    folder.
    """
    from typing import List, Tuple

    import numpy as np

    from spins.invdes import problem_graph
    from spins.invdes.problem_graph import optplan

    # Yee cell grid spacing in nanometers.
    GRID_SPACING = 40
    # If `True`, perform the simulation in 2D. Else in 3D.
    SIM_2D = True
    # Silicon refractive index to use for 2D simulations. This should be the
    # effective index value.
    SI_2D_INDEX = 2.20
    # Silicon refractive index to use for 3D simulations.
    SI_3D_INDEX = 3.45


    def main() -> None:
        """Runs the optimization."""
        # Create the simulation space using the GDS files.
        sim_space = create_sim_space("sim_fg.gds", "sim_bg.gds")

        # Setup the objectives and all values that should be recorded (monitors).
        obj, monitors = create_objective(sim_space)

        # Create the list of operations that should be performed during
        # optimization. In this case, we use a series of continuous parametrizations
        # that approximate a discrete structure.
        trans_list = create_transformations(
            obj, monitors, sim_space, cont_iters=100, min_feature=100)

        # Execute the optimization and indicate that the current folder (".") is
        # the project folder. The project folder is the root folder for any
        # auxiliary files (e.g. GDS files). By default, all log files produced
        # during the optimization are also saved here. This can be changed by
        # passing in a third optional argument.
        plan = optplan.OptimizationPlan(transformations=trans_list)
```

```
        problem_graph.run_plan(plan, ".")


    def create_sim_space(gds_fg: str, gds_bg: str) -> optplan.SimulationSpace:
        """Creates the simulation space.

        The simulation space contains information about the boundary conditions,
        gridding, and design region of the simulation. The material stack is
        220 nm of silicon surrounded by oxide. The refractive index of the silicon
        changes based on whether the global viarble `SIM_2D` is set.

        Args:
            gds_fg: Location of the foreground GDS file.
            gds_bg: Location of the background GDS file.

        Returns:
            A `SimulationSpace` description.
        """
        mat_oxide = optplan.Material(index=optplan.ComplexNumber(real=1.5))
        if SIM_2D:
            device_index = SI_2D_INDEX
        else:
            device_index = SI_3D_INDEX

        mat_stack = optplan.GdsMaterialStack(
            background=mat_oxide,
            stack=[
                optplan.GdsMaterialStackLayer(
                    foreground=mat_oxide,
                    background=mat_oxide,
                    gds_layer=[101, 0],
                    extents=[-10000, -110],
                ),
                optplan.GdsMaterialStackLayer(
                    foreground=optplan.Material(
                        index=optplan.ComplexNumber(real=device_index)),
                    background=mat_oxide,
                    gds_layer=[100, 0],
                    extents=[-110, 110],
                ),
            ],
        )

        if SIM_2D:
            # If the simulation is 2D, then we just take a slice through the
            # device layer at z = 0. We apply periodic boundary conditions along
            # the z-axis by setting PML thicknes to zero.
            sim_region = optplan.Box3d(
                center=[0, 0, 0], extents=[5000, 5000, GRID_SPACING])
            pml_thickness = [10, 10, 10, 10, 0, 0]
        else:
            sim_region = optplan.Box3d(center=[0, 0, 0], extents=[5000, 5000, 2000])
            pml_thickness = [10, 10, 10, 10, 10, 10]

        return optplan.SimulationSpace(
            name="simspace_cont",
            mesh=optplan.UniformMesh(dx=GRID_SPACING),
            eps_fg=optplan.GdsEps(gds=gds_fg, mat_stack=mat_stack),
```

```
            eps_bg=optplan.GdsEps(gds=gds_bg, mat_stack=mat_stack),
            sim_region=sim_region,
            selection_matrix_type="direct_lattice",
            boundary_conditions=[optplan.BlochBoundary()] * 6,
            pml_thickness=pml_thickness,
        )


    def create_objective(sim_space: optplan.SimulationSpace
                         ) -> Tuple[optplan.Function, List[optplan.Monitor]]:
        """Creates the objective function to be minimized.

        The objective is `(1 - p1300)^2 + (1 - p1550)^2` where `p1300` and `p1500`
        is the power going from the input port to the corresponding output port
        at 1300 nm and 1500 nm. Note that in an actual device, one should also add
        terms corresponding to the rejection modes as well.

        Args:
            sim_space: Simulation space to use.

        Returns:
            A tuple `(obj, monitors)` where `obj` is a description of objective
            function and `monitors` is a list of values to monitor (save) during
            the optimization process.
        """
        # Create the waveguide source at the input.
        wg_source = optplan.WaveguideModeSource(
            center=[-1770, 0, 0],
            extents=[GRID_SPACING, 1500, 600],
            normal=[1, 0, 0],
            mode_num=0,
            power=1.0,
        )
        # Create modal overlaps at the two output waveguides.
        overlap_1550 = optplan.WaveguideModeOverlap(
            center=[1730, -500, 0],
            extents=[GRID_SPACING, 1500, 600],
            mode_num=0,
            normal=[1, 0, 0],
            power=1.0,
        )
        overlap_1300 = optplan.WaveguideModeOverlap(
            center=[1730, 500, 0],
            extents=[GRID_SPACING, 1500, 600],
            mode_num=0,
            normal=[1, 0, 0],
            power=1.0,
        )

        power_objs = []
        # Keep track of metrics and fields that we want to monitor.
        monitor_list = []
        for wlen, overlap in zip([1300, 1550], [overlap_1300, overlap_1550]):
            epsilon = optplan.Epsilon(
                simulation_space=sim_space,
                wavelength=wlen,
            )
            sim = optplan.FdfdSimulation(
```

```
                source=wg_source,
                # Use a direct matrix solver (e.g. LU-factorization) on CPU for
                # 2D simulations and the GPU Maxwell solver for 3D.
                solver="local_direct" if SIM_2D else "maxwell_cg",
                wavelength=wlen,
                simulation_space=sim_space,
                epsilon=epsilon,
            )
            # Take a field slice through the z=0 plane to save each iteration.
            monitor_list.append(
                optplan.FieldMonitor(
                    name="field{}".format(wlen),
                    function=sim,
                    normal=[0, 0, 1],
                    center=[0, 0, 0],
                ))
            if wlen == 1300:
                # Only save the permittivity at 1300 nm because the permittivity
                # at 1550 nm is the same (as a constant permittivity value was
                # selected in the simulation space creation process).
                monitor_list.append(
                    optplan.FieldMonitor(
                        name="epsilon",
                        function=epsilon,
                        normal=[0, 0, 1],
                        center=[0, 0, 0]))

            overlap = optplan.Overlap(simulation=sim, overlap=overlap)

            power = optplan.abs(overlap)**2
            power_objs.append(power)
            monitor_list.append(
                optplan.SimpleMonitor(name="power{}".format(wlen), function=power))

        # Spins minimizes the objective function, so to make `power` maximized,
        # we minimize `1 - power`.
        obj = 0
        for power in power_objs:
            obj += (1 - power)**2

        monitor_list.append(optplan.SimpleMonitor(name="objective", function=obj))

        return obj, monitor_list


    def create_transformations(
            obj: optplan.Function,
            monitors: List[optplan.Monitor],
            sim_space: optplan.SimulationSpaceBase,
            cont_iters: int,
            num_stages: int = 3,
            min_feature: float = 100,
    ) -> List[optplan.Transformation]:
        """Creates a list of transformations for the device optimization.

        The transformations dictate the sequence of steps used to optimize the
        device. The optimization uses `num_stages` of continuous optimization. For
        each stage, the "discreteness" of the structure is increased (through
```

```
        controlling a parameter of a sigmoid function).

        Args:
            opt: The objective function to minimize.
            monitors: List of monitors to keep track of.
            sim_space: Simulation space ot use.
            cont_iters: Number of iterations to run in continuous optimization
                total acorss all stages.
            num_stages: Number of continuous stages to run. The more stages that
                are run, the more discrete the structure will become.
            min_feature: Minimum feature size in nanometers.

        Returns:
            A list of transformations.
        """
        # Setup empty transformation list.
        trans_list = []

        # First do continuous relaxation optimization.
        # This is done through cubic interpolation and then applying a sigmoid
        # function.
        param = optplan.CubicParametrization(
            # Specify the coarseness of the cubic interpolation points in terms
            # of number of Yee cells. Feature size is approximated by having
            # control points on the order of `min_feature / GRID_SPACING`.
            undersample=3.5 * min_feature / GRID_SPACING,
            simulation_space=sim_space,
            init_method=optplan.UniformInitializer(min_val=0.6, max_val=0.9),
        )

        iters = max(cont_iters // num_stages, 1)
        for stage in range(num_stages):
            trans_list.append(
                optplan.Transformation(
                    name="opt_cont{}".format(stage),
                    parametrization=param,
                    transformation=optplan.ScipyOptimizerTransformation(
                        optimizer="L-BFGS-B",
                        objective=obj,
                        monitor_lists=optplan.ScipyOptimizerMonitorList(
                            callback_monitors=monitors,
                            start_monitors=monitors,
                            end_monitors=monitors),
                        optimization_options=optplan.ScipyOptimizerOptions(
                            maxiter=iters),
                    ),
                ))

            if stage < num_stages - 1:
                # Make the structure more discrete.
                trans_list.append(
                    optplan.Transformation(
                        name="sigmoid_change{}".format(stage),
                        parametrization=param,
                        # The larger the sigmoid strength value, the more "discrete"
                        # structure will be.
                        transformation=optplan.CubicParamSigmoidStrength(
                            value=4 * (stage + 1)),
```

```
                    ))
        return trans_list


    if __name__ == "__main__":
        main()
```