spins Documentation

Release 0.2.0

Vuckovic Lab

Jan 21, 2022

Contents:

CHAPTER 1

Introduction

SPINS-B is the open source version of SPINS, a framework for gradient-based (adjoint) photonic optimization developed over the past decade at Jelena Vuckovic's Nanoscale and Quantum Photonics Lab at Stanford University. For commercial use, the full version can be licensed through the Stanford Office of Technology and Licensing (see FAQ).

The overall architecture is explained in our paper Nanophotonic Inverse Design with SPINS: Software Architecture and Practical Considerations.

1.1 Features

- Gradient-based (adjoint) optimization of photonic devices
- 2D and 3D device optimization using finite-difference frequency-domain (FDFD)
- Support for custom objective functions, sources, and optimization methods
- Automatically save design methodology and all hyperparameters used in optimization for reproducibility

1.2 Upcoming Features

We are protoyping the next version of SPINS, known as *Goos*. This version of SPINS will support these new features:

- Integration with FDTD solvers
- Co-optimization of multiple device regions simulataneously
- Easier to use and extend

1.3 Overview

Traditional nanophotonic design typically relies on parameter sweeps, which are expensive both in terms of computation power and time, and restrictive in their parameter space. Likewise, completely blackbox optimization algorithms,

such as particle swarm and genetic algorithms, are also highly inefficient. In both these cases, the computational costs limit the degrees of the freedom of the design to be quite small. In contrast, by leveraging gradient-based optimization methods, our nanophotonic inverse design algorithms can efficiently optimize structures with tens of thousands of degrees of freedom. This enables the algorithms to explore a much larger space of structures and therefore design devices with higher efficiencies, smaller footprint, and novel functionalities.

1.4 Requirements

- Python 3.5+
- Some version of BLAS (e.g. OpenBLAS, ATLAS, Intel MKL)
- Maxwell solver for 3D simulations

1.5 Recommendations

- We recommend using virtual environments to isolate installation from the rest of the system.
- If using OpenBLAS, we recommend setting the number of OpenBLAS threads (`OPENBLAS_NUM_THREADS` flag) to 1 as SPINS-B leverages parallelism itself.

1.6 Installation

Simply clone the SPINS-B repository and run `pip`:

```
$ pip3 install ./spins-b
```

1.7 Getting Started

See the grating coupler optimization example and the wavelength demultiplexer example in the `examples` folder. The grating coupler example covers setting up, running, and resuming a 2D optimization. The wavelength demultiplexer example covers setting up and running a 3D optimization as well as various ways of processing the optimization logs.

More documentation is forthcoming.

1.8 General Concepts

- **Optimization plan**: The optimization plan defines all the photonic optimization problem (i.e. simulation region and desired objective) as well as the sequence of optimization steps to achieve that objective. You define an optimization plan which is then executed by SPINS-B. Doing so enables you to have an exact record of all the parameters used to design a device as well as the ability to resume optimization if the optimization fails midway.
- **Simulation space**: The simulation space defines the simulation region as well as the design region (see below).
- **Design area and design region**: The design region is the region of the permittivity distribution that is allowed to vary during the optimization. The design region is defined as the difference between two permittivity distributions: Where the difference is non-zero corresponds to the design region. Since most photonic devices are fabricated using top-down lithography, SPINS-B by default (this can be changed) assumes that the permittivity distribution along the z-axis is the same, and hence we speak of a *design area*.

- **Parametrization**: The parametrization defines how to describe the permittivity of the design area. The simplest parametrization is to simply describe the value of each pixel on the Yee grid.
- **Monitors**: Monitors are used to log data during the optimization process. *Simple monitors* simply record the value of a function whereas *field monitors* post-processes vector field data and can select out a particular plane to save data.
- **Transformation**: Optimization in SPINS-B actually consists of a sequence of optimization problems. Each optimization is described by a *transformation* (because they transform the parametrization from one to another).

1.9 FAQ

1.9.1 What's different between SPINS-B and SPINS?

SPINS is a fully-featured optimization design suite available for commercial use. It is a superset of SPINS-B and includes the ability to design devices without writing any code with user-friendly interfaces and to apply precise fabrication constraints (minimum gap and curvature constraints). All devices shown in our published work rely on capabilities found in the fully-featured SPINS.

1.9.2 How are structures simulated?

SPINS-B uses the finite difference frequency domain (FDFD) simulation method. This choice was made because in many photonic device designs, we are concerned with device operation in a small bandwidth at particular frequencies. The FDFD method is often faster than the more widely used finite difference time domain (FDTD) method in these cases.

SPINS-B can use both a CPU-based solver or the GPU-accelerated Maxwell FDFD solver. For 2D simulations, we recommend using a direct matrix CPU-based solver ("local_direct") because it is faster. 3D simulations require too much memory and an iterative solver must be used. We recommend the GPU-accelerated MaxwellFDFD solver ("maxwell_cg") in this case.

1.10 Publications

Any publications resulting from the use of this software should acknowledge SPINS-B and cite the following papers:

For general device optimization:

- Su et al. Nanophotonic Inverse Design with SPINS: Software Architecture and Practical Considerations. *arXiv:1910.04829* (2019).

For grating coupler optimization:

- Su et al. Fully-automated optimization of grating couplers. *Opt. Express* (2018).
- Sapra et al. Inverse design and demonstration of broadband grating couplers. *IEEE J. Sel. Quant. Elec.* (2019).

CHAPTER 2

Navigating SPINS Codebase

SPINS is a large nanophotonic optimization library with several different structures. Here we provide a brief overview of the sublibraries in SPINS:

- `invdes`: This is the initially-released library that handles the inverse design aspect of SPINS.
- `goos`: This is the next generation inverse design library meant to replace the older `invdes` library. This library is still under development, though you can run many optimizations already with `goos`.
- `goos_sim`: This module contains simulators for `goos`.
- `fdfd_tools` and `fdfd_solvers`: This sublibraries handle setup and running finite-difference frequency-domain (FDFD) simulations.
- `gridlock`: This library handles drawing on the Yee grid. The Yee grid is a set of offset grids that are used in FDTD and FDFD simulations.

There are a few other small sub-libraries under the `spins` package that perform tasks specific for the `invdes` library, which we will ignore here.

CHAPTER 3

Invdes

3.1 Examples

3.1.1 Example: Grating Coupler Optimization

Quick start guide

Running grating optimization

To run the example grating coupler optimization, execute the following:

```
python3 grating.py run save-folder-name
```

The example provided has the following parameters

Material stack parameters:

- Oxide cladding
- 220 nm silicon
- 2 um buried-oxide (BOX) layer
- Silicon substrate

Grating parameters:

- Grating length = 12 um
- Partial etch depth = 0.5 (1 would be fully etched)

Source parameters:

- 1550 nm
- 10.4 um mode-field-diameter
- 10 degrees angle of incidence

Simulation/optimization parameters:

- Grid discretization = 40 nm
- Number of PML layers = 10
- Minimum feature size = 100 nm
- 60 iterations of continuous optimization
- 200 iterations of discrete optimization

View results

To generate the results, run the following:

```
python3 grating.py view save-folder-name
```

To get text-only output , run `view_quick` instead:

```
python3 grating.py view_quick save-folder-name
```

Generate GDS

A GDS file (named `grating.gds`) is automatically generated in the `save-folder-name` at the end of an optimization.

We generate a 2d design by extruding the 1D optimized grating coupler design. In the example file the extrude length is 12 um. To generate this GDS we run:

```
python3 grating.py gen_gds save-folder-name
```

Resume optimization

If for some reason an optimization is terminated, it can be resumed by running:

```
python3 grating.py resume save-folder-name
```

Modifying grating coupler parameters

For the sake of an example, let's adjust the optimization for a 60% partial etch grating coupler in 300 nm thick silicon nitride on 3um of buried oxide layer with air-cladding. We'll also adjust the source to be a normally incident at 1300 nm.

Material stack

In the `run_opt` function we find parameters for waveguide thickness (`wg_thickness`), thickness of buried-oxide layer (`box_thickness`), and partial etch fraction (`etch_frac`), which we can adjust for our silicon nitride example.

```
wg_thickness = 300

sim_space = create_sim_space(
    "sim_fg.gds",
    "sim_bg.gds",
    grating_len=grating_len,
    box_thickness=3000,
    wg_thickness=wg_thickness,
    etch_frac=0.6,
    wg_width=wg_width)
```

Next, to adjust the material properties of the stack we look in the `create_sim_space` function and find where the stack variable is defined.

The substrate and buried oxide layer are first set:

```
stack = [
    optplan.GdsMaterialStackLayer(
        foreground=optplan.Material(mat_name="Si"),
        background=optplan.Material(mat_name="Si"),
        # Note that layer number here does not actually matter because
        # the foreground and background are the same material.
        gds_layer=[300, 0],
        extents=[-10000, -box_thickness],
    ),
    optplan.GdsMaterialStackLayer(
        foreground=optplan.Material(mat_name="SiO2"),
        background=optplan.Material(mat_name="SiO2"),
        gds_layer=[300, 0],
        extents=[-box_thickness, 0],
    ),
]
```

and so adjusting the `box_thickness` earlier is the only change we need to make. As for the grating coupler, we look at the elements appended to this stack array below. Pre-defined materials in Spins-B are `"Air"`, `"SiO2"`, `"Si"`, `"Si3N4"`. For greatest generality, we'll define a custom material for the silicon nitride in this example where we set the real part of the index to be 2.0 and the imaginary (loss) to be 0.0.

Note: In addition to specifying a single refractive index value, a custom material can be added as well which interpolates dispersion from provided data. Reference `optplan.Material` for more information.

```
# If `etch-frac` is 1, then we do not need two separate layers.
if etch_frac != 1:
    stack.append(
        optplan.GdsMaterialStackLayer(
            foreground=optplan.Material(index=optplan.ComplexNumber(real=2.0,imag=0.
↪0))
            background=optplan.Material(mat_name="Air"),
            gds_layer=[LAYER_SILICON_NONETCHED, 0],
            extents=[0, wg_thickness * (1 - etch_frac)],
        ))
stack.append(
    optplan.GdsMaterialStackLayer(
        foreground=optplan.Material(index=optplan.ComplexNumber(real=2.0,imag=0.0))
        background=optplan.Material(mat_name="Air"),
```

(continues on next page)

(continued from previous page)

```
        gds_layer=[LAYER_SILICON_ETCHED, 0],
        extents=[wg_thickness * (1 - etch_frac), wg_thickness],
    ))
```

In addition, we change the background material to be `"Air"` as our grating is air-cladded.

```
mat_stack = optplan.GdsMaterialStack(
    # Any region of the simulation that is not specified is filled with
    # oxide.
    background=optplan.Material(mat_name="Air"),
    stack=stack,
)
```

Note: You can set the `visualize` flag in the `create_sim_space` function to `True` to visualize the material stack to ensure it has been built correctly.

Grating parameters

We set the partial etch depth earlier, but to re-iterate, we can adjust this value in the `run_opt` function in the arguments to the `create_sim_space` call:

```
sim_space = create_sim_space(
        "sim_fg.gds",
        "sim_bg.gds",
        grating_len=grating_len,
        box_thickness=3000,
        wg_thickness=wg_thickness,
        etch_frac=0.6,
        wg_width=wg_width)
```

We see reference to `grating_len` here, and accordingly this variable can be adjusted as well. This is set at the bottom of the example file in the `__main__` function call:

```
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument(
        "action",
        choices=("run", "view", "view_quick", "resume", "gen_gds"),
        help="Must be either \"run\" to run an optimization, \"view\" to "
        "view the results, \"resume\" to resume an optimization, or "
        "\"gen_gds\" to generate the grating GDS file.")
    parser.add_argument(
        "save_folder", help="Folder containing optimization logs.")

    grating_len = 12000
    wg_width = 12000
```

Source parameters

Source details are defined in the function `create_objective`

In this function, wavelength is set by simply adjusting the `wlen` variable in the `create_objective` function.

```
wlen = 1300
```

Note: Another location where wavelength is referenced is for plotting the permitvitty for visualization. If desired, adjust the wavelength argument in the `create_sim_space` function at the bottom:

```
if visualize:
    # To visualize permittivity distribution, we actually have to
    # construct the simulation space object.
    import matplotlib.pyplot as plt
    from spins.invdes.problem_graph.simspace import get_fg_and_bg

    context = workspace.Workspace()
    eps_fg, eps_bg = get_fg_and_bg(context.get_object(simspace), wlen=1550)
```

and then geometric properties of the beam are set by modifying the `GaussianSource` argument in the sim object:

```
sim = optplan.FdfdSimulation(
    source=optplan.GaussianSource(
        polarization_angle=0,
        theta=np.deg2rad(0),
        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,
    ),
    solver="local_direct",
    wavelength=wlen,
    simulation_space=sim_space,
    epsilon=epsilon,
)
```

For this modification, the only change we want is normal incidence (`theta = np.deg2rad(0)`). However, here we can also change the beam-width by adjusting the w0 parameter. Note,:code:‘ w0‘ is separate from `extents`, where the former is the beam radius and the latter is the extent over which the source is defined.

Note: The code supports arbitrary rotation of the source. With `psi = np.pi/2` and `polarization_angle = 0`, the polarization is set to be parallel to the grating lines and `theta` controls the angle of incidence.

Optimization parameters

Optimization parameters are set in the `create_transformation` function with the following behavior:

```
def create_transformations(
        obj: optplan.Function,
        monitors: List[optplan.Monitor],
        cont_iters: int,
        disc_iters: int,
```

(continues on next page)

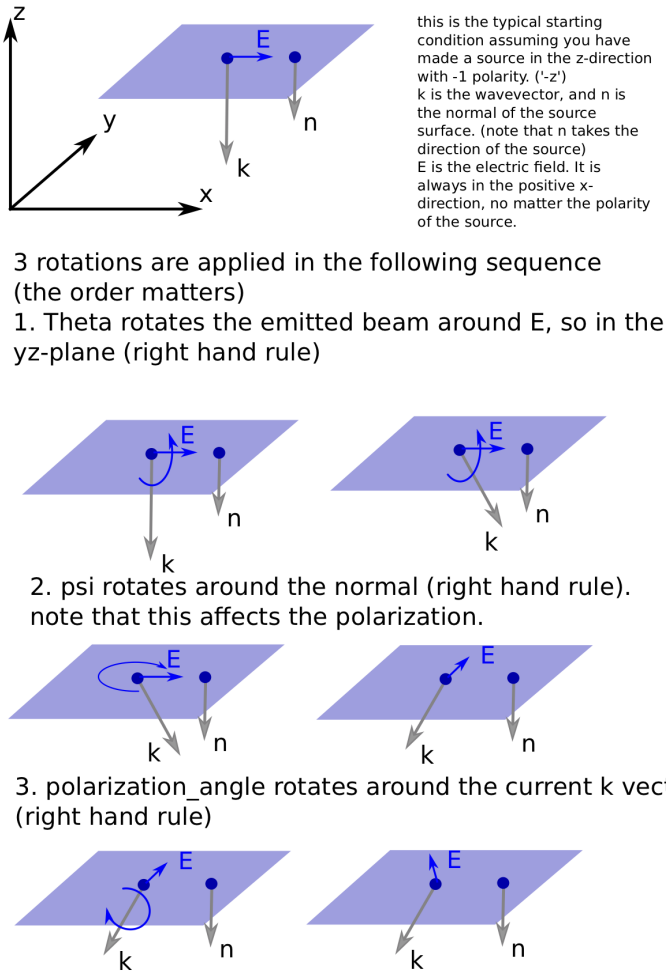

Fig. 1: Explanation of source angle rotation parameters.

(continued from previous page)

```
        sim_space: optplan.SimulationSpaceBase,
        min_feature: float = 100,
        cont_to_disc_factor: float = 1.1,
) -> List[optplan.Transformation]:
```

Accordingly, to change the number of continuous or discrete optimzation iterations we adjust this argument where this function is called in the run_opt function:

```
trans_list = create_transformations(
    obj, monitors, cont_iters=60, disc_iters=200, sim_space, min_feature=100)
```

Likewise, the minimum feature size in the optimization is set here as well.

note:

```
Spins-B utilizes continuous relaxation in optimization. This means that there is a␣
↪first stage of optimization where the device permittivity is allowed to vary␣
↪continuously between the material/cladding value. This final result of this stage␣
↪acts as a seed for the discrete optimization. In this second stage, a fabricable␣
↪design is produced. In our experience, 100 iterations for each stage is sufficient␣
↪to reach a local minima.
```

Additional information

Generating GDS

Once an optimization has completed in the discretization stage, a GDS file can be generated by running:

```
python3 grating.py gen_gds save-folder-name
```

The 1D optimized design is simply extruded to provide a 2D design. The extrude length is determined by the `wg_width` variable set in the `__main__` function:

```
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument(
        "action",
        choices=("run", "view", "view_quick", "resume", "gen_gds"),
        help="Must be either \"run\" to run an optimization, \"view\" to "
        "view the results, \"resume\" to resume an optimization, or "
        "\"gen_gds\" to generate the grating GDS file.")
    parser.add_argument(
        "save_folder", help="Folder containing optimization logs.")

    grating_len = 12000
    wg_width = 12000
```

Minimizing back reflections

Minimizing back reflections is set by simply turning on the flag at the beginning of the example file:

```
# If `True`, also minimize the back-reflection.
MINIMIZE_BACKREFLECTION = True
```

Setting this flag to `True` activates:

```
refl_sim = optplan.FdfdSimulation(
    source=optplan.WaveguideModeSource(
        center=wg_overlap.center,
        extents=wg_overlap.extents,
        mode_num=0,
        normal=[1, 0, 0],
        power=1.0,
    ),
    solver="local_direct",
    wavelength=wlen,
    simulation_space=sim_space,
    epsilon=epsilon,
)
refl_power = optplan.abs(
    optplan.Overlap(simulation=refl_sim, overlap=wg_overlap))**2
monitor_list.append(
    optplan.SimpleMonitor(name="mon_refl_power", function=refl_power))

# We now have two sub-objectives: Maximize transmission and minimize
# back-reflection, so we must an objective that defines the appropriate
# tradeoff between transmission and back-reflection. Here, we choose the
# simplest objective to do this, but you can use Spins-B functions to
# design more elaborate objectives.
obj = (1 - power) + 4 * refl_power
```

We see that we create an additional simulation object which performs the simulation for `WaveguideModeSource` instead of the `GaussianSource` from before. We then add the overlap monitor for the reflected power, `refl_power` with the `power` monitor for transmission to form the complete objective function, `obj`.

Note: The coefficient on `4 * refl_power` is a value that we found worked for our test example; however this is a meta-parameter that must be set for specific problems. Setting the value to `4` may be a good starting point, and tweaked based on desired performance.

Foreground and background GDS files

Documentation coming

Broadband optimization

In development

CHAPTER 4

Goos

4.1 Introduction

Goos is the second version of SPINS that offers more features with a simpler user interface, including:

- Integration with FDTD solvers
- Optimization with multiple design regions simulataneously

We encourage you to try and use Goos and provide us feedback. Goos development is still in its early phases so the interface is expected to change quickly. Documentation is still in progress as well. Unfortunately, Goos is not fully compatible with the former Invdes library, but it should not take much effort to migrate to Goos.

4.2 Installation

4.2.1 Requirements

- Python 3.6+
- Installation of a supported electromagnetic solver.

4.2.2 Installation

Simply download and install SPINS with `pip`, though we recommend using virtual environments to isolate your installation from your system.

```
$ git clone http://github.com/stanfordnqp/spins-b
$ pip install ./spins-b
```

4.2.3 Solvers

SPINS is a framework for encoding and running optimization for electromagnetics. Although SPINS comes packaged with a finite-difference frequency-domain solver (FDFD) solver, you have a choice to install other simulators in the backend. The packaged FDFD solver is only efficient for small problems, so it is recommended to install another simulator for large 2D problems or 3D problems. You must follow the installation procedure for the apprioriate simulator before using Goos. Listed below are the currently supported simulators.

FDFD Local Matrix Solver

This is a CPU-based solver that comes packaged with SPINS. This solver simply sets up the appropriate matrix equation and runs a direct matrix solve using BLAS to solve. It is therefore fast and efficient for small 2D problems but is slow for large 2D or 3D problems.

There are no additional required installation steps to use this solver but we recommend installing UMFPACK and using either ATLAS, OpenBLAS, or Intel MKL. We find that UMFPACK runs orders of magnitude faster than SuperLU, but UMFPACK will limit your simulation size to 4 GB.

On Ubuntu, simply install both SWIG and libsuitesparse:

```
$ sudo apt install libsuitesparse-dev swig
```

Then install the Python package for UMFPACK:

```
$ pip install scikit-umfpack
```

Maxwell-B

Maxwell-B is a multi-GPU finite-difference frequency-domain (FDFD) solver. As a frequency domain solver, it is efficient for problems where you care only about a few wavelengths. The Maxwell-B solver must be installed on a machine with at least one NVIDIA GPU. See the `README` file under the Maxwell source folder for details.

MEEP

MEEP is an open-source finite-difference time-domain (FDTD) solver. As a time-domain solver, it can simulate the frequency response across a broad range of frequencies with a single simulation. MEEP can be parallelized across multiple CPUs using MPI.

As of this writing, install MEEP using the nightly build as the main release contains a bug that was only recently fixed.

Custom

If the above choices are not to your liking, you may choose to define your own solver.

4.3 Core Concepts

Here we describe the core concepts behind Goos.

4.3.1 Optimization Plan

The *optimization plan* is the central object in SPINS. It consists of two parts: *nodes* that form the *problem graph* and *actions* that use the problem graph to update the state of the optimization. The problem graph provides a complete description of the entire problem, from details of the simulation to the exact objective functions. A sequence of actions define the optimization strategy. Actions may change values of variables, minimize a particular objective function, perform a discretization optimization, and so forth. Actions use the problem graph to compute any quantities, such as the objective function.

Optimization Plan Example

Consider a simple optimization plan to minimize the function $(x-1)^2$:

```
with goos.OptimizationPlan() as plan:
    x = goos.Variable(3, name="x")
    obj = (x - 1)**2

    goos.opt.scipy_minimize(obj, method="L-BFGS-B", max_iters=30)
    plan.run()
```

In this example, `x` and `obj` are nodes defined in the problem graph and `goos.opt.scipy_minimize` creates an optimization action. Note that the expressions `(x-1)**2` implicitly defines several nodes and is a shorthand for the following:

```
# The following is identical in behavior to `(x - 1)**2`.
obj = goos.Power(goos.Sum([x, goos.Constant(-1)]), 2)
```

In goos, by using the optimization plan with a `with` statement, nodes and actions that are defined within the `with`-block are added automatically to the enclosing optimization plan. The above example could alternatively have been written as follows:

```
plan = goos.OptimizationPlan()
x = goos.Variable(3, name="x")
obj = (x - 1)**2
plan.add_node(obj)

opt_action = goos.opt.ScipyOptimizer(obj, method="L-BFGS-B", max_iters=30)
plan.add_action(opt_action)

plan.run()
```

Notice that we do not have to add `x`, `obj`, and the implicit sum node explicitly; `add_node` will automatically add all the dependencies of a given node into the optimization plan.

For the rest of this documentation, we will assume that you are using context managers (`with`-block) with the optimization plan, but you can always choose to call the underlying optimization plan function to achieve the same results.

Executing Plans

When an optimization plan is defined, no code is actually executed. This means that you can write out your entire optimization methodology and make sure it is syntactically correct. Additionally, goos has some basic type checks to make sure you do not accidentally make a mistake in passing arguments to goos node. That way, you do not run a long-running optimization, only to have it suddenly fail because of a silly typo.

To actually execute a plan, you call `run`. All the actions will be executed up to that point in time. If you add additional actions afterwards, you can call `run` again to execute them:

```
with goos.OptimizationPlan() as plan:
   x = goos.Variable(3)

   # Add an action to set the variable to 5.
   # After this call, `x` still is 3.
   x.set(5)

   # After this call, `x` will be 5.
   plan.run()

   # Add another action.
   x.set(6)
   # After this call, `x` will be 6.
   plan.run()
```

Retrieving Values

You can retrieve the value of any node by calling `get` and the gradient of numerical nodes with respect to any node by calling `get_grad`:

```
with goos.OptimizationPlan() as plan:
  x = goos.Variable(3)
  y = 5 * x + 2

  x.get()  # Returns `goos.NumericFlow(array=3)`.
  y.get()  # Returns `goos.NumericFlow(array=17)`.

  y.get_grad([x])  # Returns `[goos.NumericFlow.Grad(array_grad=5)]`.
```

Note that the return value of `get` and `get_grad` are *flows*. The reason is that some nodes do not return numeric values but instead return other types, such as shapes. For numeric functions and variables, they always return `goos.NumericFlow` for the function and `goos.NumericFlow.Grad` for the gradient.

Keep in mind that calling `get` does not execute actions. It evaluates the node with the current optimization plan state. You can pass `run=True` to `get` in order to call `run` on the plan before retrieving the value:

```
with goos.OptimizationPlan() as plan:
  x = goos.Variable(3)

  x.set(5)
  x.get()  # Returns `goos.NumericFlow(array=3)`.

  # The following is equivalent to the lines:
  #     plan.run()
  #     x.get()
  x.get(run=True)  # Returns `goos.NumericFlow(array=5)`.
```

Logging and Checkpoints

Some actions, such as optimizations, will generate logging information and periodically save the state of the optimization plan. This data will be saved in the optimization plan save directory, which is set by passing in `save_path` when creating an optimization plan:

```
with goos.OptimizationPlan(save_path="/path/to/myplan") as plan:
    x = goos.Variable(3)
    obj = (x - 1)**2
    goos.opt.scipy_minimize((x - 1)**2, "CG", monitor_list=[obj])
    plan.run()

    # You should see `/path/to/myplan` contain Pickle files containing the
    # state of each optimization step. Each file contains information
    # about `x` as well as the objective function value `obj`.
```

Instead of relying on actions to save the state, you can force *checkpoints* to be saved at any time:

```
with goos.OptimizationPlan(save_path="/path/to/myplan") as plan:
    x = goos.Variable(3)
    # The following will write the state to
    # `/path/to/myplan/mycheckpoint.chkpt`.
    plan.write_checkpoint("mycheckpoint.chkpt")
```

Saving and Loading Plans

Optimization plans can be loaded and saved using the `load` and `save` commands. Note that these functions only load and save the problem graph and actions but do not save any variable state. See *Logging and Checkpoints* for saving actual state.

```
with goos.OptimizationPlan(save_path="/path/to/myplan") as plan:
    x = goos.Variable(3, name="x")
    goos.opt.scipy_minimize((x - 1)**2, "CG")
    # The following creates `/path/to/myplan/optplan.json`.
    plan.save()


with goos.OptimizationPlan(save_path="/path/to/myplan") as plan:
    # The following loads from `/path/to/myplan/optplan.json`.
    # You could also explicitly state the save folder:
    # `plan.load("/path/to/myplan")`.
    plan.load()
    x = plan.get_node("x")
    # `x.get() == 3`

    plan.run()
    # `x.get() == 1`.
```

Debugging Plans

Because plans are not executed as soon as nodes are declared, you may find it useful to declare temporary debugging plans to test out the behavior of your code:

```
with goos.OptimizationPlan() as plan:

    # Some code that involves a lot of computation (e.g. electromagnetic
    # simulations).
    ...

    x = goos.Variable(3)
```

(continues on next page)

(continued from previous page)

```
    y = x**2

    x.set(3)

    # We want to know what the value of `y` would be here but we do not
    # want to run the plan and trigger the code that involves a lot of
    # computation. Instead, we create a temporary plan which only includes
    # `x` and `y`.
    with goos.OptimizationPlan() as temp_plan:
        # This new plan does NOT have the `x.set` action so we repeat it.
        x.set(3)
        # This executes the `temp_plan` and not the original `plan`.
        y.get(run=True)  # Returns 9.

        # We could continue to test out what `y` equals with different
        # values of `x`.
        x.get(4)
        y.get(run=True)

    # Here, `temp_plan` is destroyed, along with all of its actions and state.
    # (`x.get()` returns 3 and `y.get()` returns 9 still).
```

4.3.2 Problem Graph

Variable Nodes

Variable nodes can be thought of as nodes that hold raw numeric data. Variables act as sources of data for the rest of the computational graph, and the state of the optimization plan is fully captured by the state of all the variables.

To create a variable, simply pass in the initial value:

```
var_scalar = goos.Variable(4)
var_vector = goos.Variable([3, 4, 5], name="vector")
```

As the optimization plan is executed, the actions will change the values of of the variables. However, you can directly set the value of a variable using the `set` method:

```
var = goos.Variable(4)
var.set(5)

# We can also use one variable to set the value of another.
var2 = goos.Variable(8)
var2.set(var)
```

In SPINS, a variable state contains the following:

- a numeric value, stored as a NumPy array
- upper and lower bounds on each element of array
- its frozen state

The upper and lower bounds of a variable (i.e. box constraints) are used by optimizers when performing an optimization. These are treated differently than generic constraints as there are many optimization algorithms that can handle these box constraints but not general constraints. Upper and lower bounds can be set during initialization:

```
# Constraint the variable to between 0 and 10.
var = goos.Variable(5, lower_bounds=0, upper_bounds=10)

# Each entry can have different bounds. The following constraints
# the first entry to be between 0 and 1 and the second entry to be between
# 0 and 2.
var2 = goos.Variable([0.1, 0.2], lower_bounds=[0, 0], upper_bounds=[1, 2])
```

Variables also have a boolean flag indicating whether they are *frozen*. Frozen variables have zero gradient and are not (usually) modified by actions. The "frozenness" can be changed by calling `freeze` and `thaw`:

```
x = goos.Variable(3)
y = goos.Variable(4)


x.freeze()
goos.opt.scipy_minimize(x**2 + y**2, "CG")
# `x` remains at 3 because it is frozen. `y` is now zero.
assert x.get() == 3
assert y.get() == 0

x.thaw()
goos.opt.scipy_minimize(x**2 + y**2, "CG")
# Now both `x` and `y` are zero.
assert x.get() == 0
assert y.get() == 0

x.freeze()
# The following will raise an exception during execution!
x.set(5)
```

Sometimes a variable is meant as a parameter and should never be optimized over. In these cases, the variable can be declared as a *parameter*. Parameters are always frozen but can be set explicitly using `set`:

```
param = goos.Variable(3, parameter=True)
y = goos.Variable(4)

goos.opt.scipy_minimize(param**2 * y**2, "CG")
# `param` does not change because it is frozen by default.
assert param.get() == 3
assert y.get() == 0

# The following does NOT raise an exception because it is initialized as
# a parameter.
param.set(3)
```

Math Nodes

Math (`goos.Function`) nodes are nodes that perform mathematical operations, such as addition, multiplication, and dot products. They take in numerical input and produce numerical output (specifically, they take other `goos.Function` nodes as input and produce `goos.NumericFlow` as output).

SPINS has implemented a common subset of useful mathematical functions and provided operator overloads for the basic operations.

```
x = goos.Variable([1, 2])
y = goos.Variable([3, 4])

# Element-wise operations.
sum_node = x + y
prod_node = x * y
sub_node = x - y
div_node = x / y
# Note that the power must be a constant. It can NOT be a variable.
power_node = x**2

# Vector operations.
# Computes `||x||`.
norm_node = goos.norm(x)
dot_prod_node = goos.dot(x, y)
```

Shape Nodes

Shape nodes are those that represent a permittivity distribution and include objects such as cylinders and boxes. Parametrized permittivity distributions are also shapes.

Simple shapes, such as cylinders, as straightforward to create:

```
box = goos.Cuboid(pos=goos.Constant([0, 0, 0]),
                  extents=goos.Constant([1000, 400, 220]),
                  material=goos.material.Material(index=2))
cyl = goos.Cylinder(pos=goos.Constant([100, 0, 0]),
                    radius=goos.Constant([50]),
                    height=goos.Constant([220]),
                    material=goos.material.Material(index=3))

# Notice that you can also compute these quantities dynamically.
start_pos = goos.Constant([0, 0, 0])
delta_pos = goos.Constant([100, 0, 0])
boxes = []
for i in range(10):
  boxes.append(goos.Cuboid(pos=start_pos + i * delta_pos,
                           extents=goos.Constant([10, 10, 10]),
                           material=goos.material.Material(index=2))
```

SPINS also defines certain shapes useful for inverse design. For example, the pixelated continuous shape represents a shape composed of voxels that can take on permittivities continuously between that of two materials. Often there are special functions defined to help create these shapes.

```
# The initializer is a function that accepts a single parameter `shape` and
# must return an array of numbers with shape `shape`.
def initializer(shape):
  return np.random.random(shape) * 0.2 + 0.5

# `vae` is a `goos.Variable` node that controls the value of the shape node
# `design`.
var, design = goos.pixelated_cont_shape(
    initializer=initializer,
    pos=goos.Constant([0, 0, 0]),
    extents=[2000, 2000, 220],
    pixel_size=[40, 40, 220],  # Each voxel is 40 x 40 x 220.
```

(continues on next page)

(continued from previous page)

```
    # The pixels can have refractive indices between 1 and 2.
    material=goos.material.Material(index=1),
    material2=goos.material.Material(index=2))
```

Simulation Nodes

Simulations are nodes themselves. Simulation nodes take as input the permittivity distribution and produces as output the electric fields and other related quantities, such as modal overlaps. Note that each simulation node has a different set of capabilities, so you should consult the documentation for each simulation node. Typically, setting up the simulation involves the following components:

- Specification of the simulation space, i.e. simultaions extents, meshing, boundary conditions, etc.
- Permittivity distribution to simulate
- Source specification
- Output specification, e.g. electric fields, modal overlaps, etc.
- Additional simulation-specific parameters.

As an example, below is how to setup a FDFD simulation using the built-in Maxwell solver:

```
# Import the desired simulator.
from spins.goos.simulator import maxwell

waveguide = goos.Cuboid(...)
var, design = goos.pixelated_cont_shape(...)

# Group the waveguide and design together into one permittivity
# distribution.
eps = goos.GroupShape([waveguide, design])

sim = maxwell.fdfd_simulation(
    name="sim",
    wavelength=1550,  # Wavelength of the simulation.
    simulation_space=maxwell.SimulationSpace(
        mesh=maxwell.UniformMesh(dx=40),
        sim_region=goos.Box3d(
            center=[0, 0, 0],
            extents=[4000, 4000, sim_z_extent],
        ),
        pml_thickness=[400, 400, 400, 400, 0, 0]),
    eps=eps,
    sources=[
        # Add a single waveguide mode source.
        maxwell.WaveguideModeSource(center=[-1400, 0, 0],
                                    extents=[0, 2500, 1000],
                                    normal=[1, 0, 0],
                                    mode_num=0,
                                    power=1)
    ],
    background=goos.material.Material(index=1.0),
    outputs=[
        maxwell.Epsilon(name="eps"),
        maxwell.ElectricField(name="field"),
        maxwell.WaveguideModeOverlap(name="overlap",
```

(continues on next page)

(continued from previous page)

```
                                    center=[0, 1400, 0],
                                    extents=[2500, 0, 1000],
                                    normal=[0, 1, 0],
                                    mode_num=0,
                                    power=1),
    ],
    solver="local_direct",
)

# We can now extract the simulation outputs either as `sim[0]`, `sim[1] `,
# etc. or as `sim["eps"]`, `sim["field"]`, etc. because we added a name
# for each output.

# Define an objective function based on the modal overlap integral.
obj = 1 - goos.abs(sim["overlap"])**2
```

Flows

Flows are data objects that are generated by *nodes*. These include general NumPy arrays and shape objects that represent permittivity distributions. Flows are essentially a generalization of tensors used in machine learning. Unless you are implementing your own nodes, you will mainly only encounter nodes when evaluating the result of a node (i.e. call `node.get`).

NumericFlow

Numeric flows have a single field called `array`, which contains a multi-dimensional NumPy array. There is no instrinsic meaning behind the values in the array. Math nodes (those that inherit from `goos.Function`) return numeric flows.

Numeric flows have some basic overloads for == so that you can quickly compare numeric flows and the underlying array.

```
x = goos.Variable(3)
y = x + 1
flow = y.get()

# Flows have an `array` property.
assert flow.array == 4
# But for simple cases, you can drop the `array`.
assert flow == 4
```

4.3.3 Actions

Where as nodes setup the problem grpah and determine how values are computed, actions actually perform the computation and are able to modify variable values. In fact, only actions are allowed to modify the state of any variables. The most common action is to run an optimization, but a single optimization plan may contain many actions, including setting/changing variable values and running a discretization procedure.

You can distinguish an action from a node in that actions always inherit from `goos.Action`, though it should be clear from context whether something is an action.

In the code snippet below, the calls to `goos.opt.scipy_minimize` and `goos.Variable.set` generate actions.

```
with goos.OptimizationPlan() as plan:
  x = goos.Variable(1)
  target = goos.Variable(3, parameter=True)
  obj = (x - target)**2

  goos.opt.scipy_minimize(obj, "L-BFGS-B", max_iters=10)

  target.set(4)
  goos.opt.scipy_minimize(obj, "L-BFGS-B", max_iters=10)
```

Remember that actions are not actually executed until `OptimizationPlan.run` is invoked. The optimization plan maintains an action pointer that remembers that last executed action, so you can execute an action, add more actions, and then call `run` again to execute only the newly added actions:

```
with goos.OptimizationPlan() as plan:
  x = goos.Variable(1)

  # Action to increment `x`.
  x.set(x + 1)

  print(x.get().array)  # Prints 1.

  plan.run()

  print(x.get().array)  # Prints 2.

  x.set(x + 1)
  plan.run()

  print(x.get().array)  # Prints 3.
```

4.4 Extending SPINS

Here we discuss how to add custom nodes, actions, and flows.

4.4.1 Custom Nodes

To create a custom node,

- Create a class that inherits from `goos.ProblemGraphNode`
- Add a `node_type` class field that uniquely identifies the class.
- Add a type-annotated constructor that marks node inputs.
- Implement `eval` and `grad`.
- For performance gains, implement `eval_const_flags`.

A basic example of a node that doubles the value of its input is shown below:

```
class DoubleNode(goos.Function):
  node_type = "mycustomnodes.double_node"

  def __init__(self, node: goos.Function) -> None:
```

(continues on next page)

(continued from previous page)

```
    super().__init__(node)

  def eval(self, input_vals: List[goos.NumericFlow]) -> goos.NumericFlow:
    return goos.NumericFlow(input_vals[0].array * 2)

  def grad(self, input_vals: List[goos.NumericFlow],
           grad_val: goos.NumericFlow.Grad) -> goos.NumericFlow.Grad:
    return goos.NumericFlow.Grad(grad_val.array_grad * 2)
```

This custom node inherits from `goos.Function`, which in turn inherits from `goos.ProblemGraphNode`. By inheriting from `goos.Function`, we enable users of our node to perform other numeric operations on it:

```
var = goos.Variable(3)
node = DoubleNode(var)
# The following line is only possible because we inherit from
# `goos.Function` rather than `goos.ProblemGraphNode` directly.
obj = node + 3
```

Next, we named our node "mycustomnodes.double_node". This needs to be a unique name across all possible nodes. We therefore recommend the format "your_own_unique_identifier.node_name". This name is used to serialize and deserialize the node from disk.

The node constructor is defined and properly type-annotated. The type annotations are used internally to construct a schema for the node, which is used to validate its inputs. For example, because the sole argument `node` is defined as a `goos.Function`, the following will raise an error:

```
# Raises an error because `3` is not a `goos.Function`!
node = DoubleNode(3)

# Instead, we need to wrap the 3 as a constant.
node = DoubleNode(goos.Constant(3))
```

You may find it annoying for users to have to wrap a constant input as a `goos.Constant` function node. For the moment, we will live with this fact; we will come back to the topic of usability later.

Because the inputs must be serializable, only certain type annotations are supported at the moment:

- The native types `int`, `float`, `str`, `bool`, and `complex`
- `numpy.ndarray`
- The composite types `List` and `Union`
- Any `goos.ProblemGraphNode`
- Any `goos.Model`

In the constructor, we call `__init__` with a single argument that marks all the problem graph node dependencies. The flows generated by all the nodes given as an argument to `__init__` will be passed to `eval` and `grad` as `input_vals`. If there is more than one dependency, the order in which they are passed to `__init__` indcates the order that they are passed to `eval` and `grad`:

```
class SumAndDoubleNode(goos.Function):
   node_type = "mycustomnodes.sum_and_double_node"

   def __init__(self, node1: goos.Function, node2: goos.Function) -> None:
     super().__init__([node1, node2])

   def eval(self, input_vals: List[goos.NumericFlow]) -> goos.NumericFlow:
```

(continues on next page)

(continued from previous page)

```
    # `input_vals` contains flows for `node1` followed by `node2`.
    node1_val = input_vals[0].array
    node2_val = input_vals[1].array
    ...
```

Finally, we implement the node logic by defining `eval` and `grad`. `eval` is called to evaluate the function and `grad` is called to evaluate the gradient. Specifically, `eval` accepts a list of input flows from the nodes marked as dependencies in the constructor and must produce a single flow as output. `grad` accepts a list of input flows as well as the current backward gradient value and produces the corresponding gradient flow.

Implementing `eval` and `grad`

`eval` and `grad` form the backbone of the backpropr algorithm to automatically compute objective function values and their gradients. Specifically, if the objective function is given by f, then the `grad` function for a node g with inputs $x_1, x_2, \cdots, x_n$ should compute the partial derivatives $\frac{df}{dx_1}, \frac{df}{dx_2}, \cdots, \frac{df}{dx_n}$. The partial derivative $\frac{df}{dg}$ is given as the second argument to `grad`.

For example, suppose we have a node that takes two inputs and implements the function $g(x, y) = x \cdot (y + 1)$:

```
class NodeG(goos.Function):
   node_type = "mycustomnodes.node_g"

   def __init__(self, node1: goos.Function, node2: goos.Function) -> None:
     super().__init__([node1, node2])

   def eval(self, input_vals: List[goos.NumericFlow]) -> goos.NumericFlow:
     x = input_vals[0].array
     y = input_vals[1].array

     return goos.NumericFlow(x * (y + 1))
```

The `grad` function needs to compute $\frac{dg}{dx} = y + 1$ and $\frac{dg}{dy} = x$ given $\frac{df}{dg}$, which is passed as the second argument in `grad`:

```
class NodeG(goos.Function):
   ...

   def grad(self, input_vals: List[goos.NumericFlow],
                  grad_val: goos.NumericFlow.Grad)
                  -> List[goos.NumericFlow.Grad]:
     x = input_vals[0].array
     y = input_vals[1].array

     df_dx = (y + 1) * grad_val.array_grad
     df_dy = x * grad_val.array_grad

     return [goos.NumericFlow.Grad(dg_dx), goos.NumericFlow.Grad(dg_dy)]
```

In order to ensure the correctness and reproduciblity of SPINS, the following rules must hold true for `eval` and `grad`:

- Flows should NOT be modified. If you want to modify a flow, make a copy first.
- Flow values should only depend on values computed from the input flows or parameters passed in through the constructor.

Note that in SPINS, the flow system uses duck typing: Anything that has the appropriate properties of a flow is considered a flow of that type. Furthermore, a flow type may be considered as more than one type of flow. For example,

the `PixelatedContShapeFlow` can be considered a `NumericFlow` because it has an `array` property as well as a `ShapeFlow` as it has all the requisite `ShapeFlow` properties. Consequently, for single input, single output nodes, it may be advisable to clone the flow rather than creating a new one:

```
def eval(self, input_vals: List[goos.NumericFlow]) -> goos.NumericFlow:
  out_flow = copy.deepycopy(input_vals[0])
  out_flow.array = ...

  return out_flow
```

This way, this node can be used for `NumericFlow` and `PixelatedContShapeFLow`: If a `NumericFlow` is passed as input, then the output is a `NumericFlow`. If a `PixelatedContShapeFlow` is passed as input, then the output is a `PixelatedContShapeFlow`.

Models

Because nodes must be serializable, we cannot pass arbitrary objects into the constructor of a node. However, it may be beneficial to pass more complex data objects. Currently, the mechanism for doing this is through the schematics Python library. For convenience, we have aliased `schematics.models.Model` to `goos.Model` and `schematics.types` to `goos.types` and slightly modified the functionality.

For convenience, we have defined a complex number type and a NumPy array type (see `goos.optplan.schema_types`). We also have implemented a few common schema types in `goos.common_schemas`.

Usability

Sometimes, it may be clumsy to define a node directly in code. In the above `DoubleNode` example, for instance, a user must wrap a constant number like 3 as a `goos.Constant` before passing it into `DoubleNode`. To mitigate these convenience and usability issues, we recommend defining functions that create nodes on behalf of the user. For example,

```
 def double_node(node: Union[goos.Function, float]):
   if not isinstance(node, goos.Function):
     node = goos.Constant(node)
   return DoubleNode(node)

node = double_node(3)
node2 = double_node(goos.Variable(5))
```

You see this kind of node creation functions throughout the codebase in order to simplify node creation.

4.4.2 Custom Actions

A custom action must do the following:

- Inherit from `goos.Action`
- Add a `node_type` class field that uniquely identifies the class.
- Add a type-annotated constructor that marks node inputs.
- Inherit `run` method that accepts an optimization plan as an argument.

From a structural point of view, defining an action is similar to defining a node except that an action inherits from `goos.Action` instead of `goos.ProblemGraphNode` and that an action implements `run` instead of `eval` and `grad`. The above discussion about defining node types and a type-annotated constructor remains the same with the

following main exception: It is unnecessary to declare any dependencies through `super().__init__`. Below we show an example of an action that adds one to a variable.

```
class AddOne(goos.Action):
   node_type = "myactions.add_one"

   def __init__(self, var: goos.Variable) -> None:
     super().__init__()
     self._var = var

   def run(self, plan: goos.OptimizationPlan) -> None:
     val = plan.get_var_value(self._var)
     plan.set_var_value(self._var, val + 1)
```

The `run` method accepts an optimization plan as input and changes the plan state. However, the `run` method in the following ways:

- Call `eval_nodes` and `eval_grad` to evaluate the values and gradients of nodes. A plan should NOT call `node.get()` and `node.get_grad()`.
- Call `set_var_value` and `get_var_value` to get and set the values of a node. Again, this should be done in lieu of `node.get()`. Note that setting the value of a frozen variable may raise an exception.
- Call `set_var_bounds` and `get_var_bounds` to change the bounds of a variable.

As a general rule of thumb, an action may always request information about the plan state but may not be able to change the state. The `run` method should NOT add or remove nodes from the graph as this has the potential to break the reproducibility of the system. The `run` method should also NOT call `plan.run()` or any method that would invoke `plan.run()` (e.g. `node.get(run=True)`).

Usability

As with nodes, we recommend defining creation function for actions. A typical creation function is as follows:

```
def add_one(*args, **kwargs) -> AddOne:
   action = AddOne(*args, **kwargs)
   goos.get_default_plan().add_action(action)
   return action
```

In this case, this function simply forwards all the arguments to the action class though more preprocessing can be done. Additionally, the action is automatically added to the default plan, obviated the need for the user to explicitly call `add_action`.

4.4.3 Custom Flows

You may wish to define a custom flow if none of the existing flows capture the correct description of the object you wish to define. This is often true for new descriptions of shapes.

Flows must follow a few rules:

- Must inherit from `goos.Flow`.
- Must be pickable (or more specifically, dillable).
- Must provide a constructor that accepts no arguments. Note that this implies that all fields should have default values.
- Must define an inner class called `Grad`. This is automatically generated if not explicitly defined.

- Must define an inner class called `ConstFlags`. This is automatically generated if not explicitly defined.

Flows *can* do the following though:

- Inherit from more than one type of flow, though you should carefully consider the ramifications.
- Contain other flows.

To define a flow, simply create a class that inherits from `Flow`:

```
class MyFlow(goos.Flow):
    myfield: bool = False
    myfield2: np.ndarray = goos.np_zero_field(3)
    myfield3: float = 3
```

By default, flows are converted into Python dataclasses and thus the dataclass syntax can be used. Default values are provided so that the automatically defined constructor requires no arguments. `goos.np_zero_field` is a utility function that creates a field with numpy array zeros (it is short for `dataclasses.field(default=factory=lambda: np.zeros(n))`. This is necessary because best coding practices dictate that we should not default initialize with an object.

This flow can now be used like so:

```
flow = MyFlow()
flow.myfield = True

flow = MyFlow(myfield2=np.array([3,4,5]))
```

Because we did not explicitly define a `Grad` and `ConstFlags` class, they were automatically generated. In auto-generated `Grad` classes, every numeric field with name `fieldname` will have an associated field `fieldname_grad` in the `Grad` class. In auto-generated `ConstFlags`, all the fields in the flow will exist in `ConstFlags` except that they all become booleans:

```
grad_flow = MyFlow.Grad()
grad_flow.myfield2_grad = np.array([3,4,5])

const_flags = MyFlow.ConstFlags(myfield=False, myfield2=True)
```

Note that automatic generation works assuming that the dataflow model is used for the class. If you choose not to define flow fields this way, you should declare your own `Grad` and `ConstFlags`.

Gradient Flow

Every flow has an associated *gradient flow* that represents the flow containing gradient information. Consequently, during forward evaluation of the nodes, flows are passed as inputs whereas during the backward evaluation, gradient flows are passed as inputs. The gradient flow associated with a flow is simply the flow name plus `.Grad`. For example, a flow called `MyFlow` would have a gradient flow named `MyFlow.Grad`.

By default, if no inner `Grad` class is defined, a gradient flow class will be automatically constructed based on the defined fields of the `Flow`. Note that the gradient flow class autogeneration assumes that the `Flow` operates as a normal dataclass. Therefore, if you do not rely on the dataclass operation of a `Flow`, you should define your own `Grad` class.

Constant Flags

In order to optimize evaluation of the computational graph, additional flags known as *const flags* for each input are passed to `eval` and `grad`. For example, a simulation node may use the fact that a `Shape` is constant to speed up the

process of drawing the permittivity distribution. Specifically, every flow must have a `ConstFlags` inner class. It is automatically generated if not defined. This class has a field for every non-constant field of the flow.

The const flags are used in the following ways:

- Marking *constant* flow fields. Constant flow fields are those that cannot change (i.e. do not depend in any way on a `Variable`).
- Marking *frozen* flow fields. Frozen flow fields are those that do not depend on any thawed `Variable`.

Thus, by definition, all constant flow fields are also frozen flow fields, but frozen flow fields need not be constant. During function or gradient evaluation, the constant flow fields and frozen flow fields are computed and stored in a separate instance of `ConstFlags`. In other words, multiple `ConstFlags` classes will be instantiated but will server different purposes.

4.5 Advanced Topics

4.5.1 Parallelization

By default, with the exception of simulations, SPINS computes the computational graph serially and does not exploit any parallelism. Simulations, on the other hand, are parallelized as much as possible by default. Because function and gradient evaluation is dominated by the simulation time, you typically do not need to change these defaults. However, if you may choose to change this behavior on any node by calling the `parallelize` method on any `ProblemGraphNode`:

```
# Parallelize computation of the node.
node.parallelize()

# Disables parallelization.
node.parallelize(False)
```

Note that turning on parallelization may actually cause a decrease in performance due to the overhead in the setup. Keep in mind that the parallelization operates by grouping together and executing in parallel operations that (1) are marked for parallel computing and (2) can be executed independently (i.e. no direct or indirect dependency between the nodes. Therefore, there may not be any true parallelization in effect if these conditons are never met during the execution of the computational graph.

4.5.2 Array Flows

An *array flow* is, in essence, a list of other flows. Array flows are used to group together multiple flows into a single flow. Working with array flows is similar to working with arrays:

```
# Array flow is created by passing an array of flows.
flow = goos.ArrayFlow([goos.NumericFlow(4), goos.ShapeFlow()])

# Use indexing to set and get nth flow.
# Prints 4.
print(flow[0].array)
flow[1].pos = np.array([3, 4, 5])

# Prints 2.
print(len(flow))
```

`ArrayFlow.Grad` works similarly:

```
# Array flow is created by passing an array of flows.
flow = goos.ArrayFlow.Grad([goos.NumericFlow.Grad(4), goos.ShapeFlow.Grad()])

# Use indexing to set and get nth flow.
# Prints 4.
print(flow[0].array_array_grad)
flow[1].pos_grad = np.array([3, 4, 5])
```

Additionally, `ArrayFlow.Grad` supports adding multiple array flows together. When doing this summation, a flow added to `None` is just the flow itself:

```
flow1 = goos.ArrayFlow.Grad([goos.NumericFlow.Grad(1),
                             goos.NumericFlow.Grad(2)])
flow2 = goos.ArrayFlow.Grad([goos.NumericFlow.Grad(3),
                             goos.NumericFlow.Grad(4)])
flow3 = goos.ArrayFlow.Grad([None, goos.NumericFlow.Grad(5)])
flow4 = goos.ArrayFlow.Grad([None, None])

flow1 + flow2 == goos.ArrayFlow.Grad([goos.NumericFlow.Grad(4),
                                      goos.NumericFlow.Grad(6)])

flow1 + flow3 == goos.ArrayFlow.Grad([goos.NumericFlow.Grad(1),
                                      goos.NumericFlow.Grad(5)])

flow1 + flow4 == flow1
```

Using `ArrayFlowOpMixin`

For any node that produces an array flow, it is recommended that the node inherits `ArrayFlowOpMixin`. This mixin overloads the indexing operator so that individual elements of the output array flow can be easily accessed. Suppose we have a node `MyNode` that produces an array flow with two elements. Then, by inheriting from `ArrayFlowOpMixin`, we can compute the sum as follows:

```
# Computes the next two elements in the Fibonacci sequence.
class FibonacciNode(goos.ArrayFlowOpMixin, goos.ProblemGraphNode):

  def __init__(self, in1: goos.Function, in2: goos.Function) -> None:
    super().__init__([in1, in2], flow_types=[goos.Function, goos.Function])
    ...

  def eval(self, inputs: List[goos.NumericFlow]) -> goos.ArrayFlow:
    fib_next = inputs[0].array + inputs[1].array
    fib_next_next = inputs[1].array + fib_next
    return goos.ArrayFlow([goos.NumericFlow(fib_next),
                           goos.NumericFlow(fib_next_next)])

  ...

node = FibonacciNode(...)
out_sum = node[0] * node[1]
```

Note that order of inheritance. Because it is a mixin, you should inherit from `ArrayFlowOpMixin` before `ProblemGraphNode` (or any other node class). Additionally, we pass an array `flow_types` to the mixin constructor. This array sets the type of node that is returned when performing the indexing operation.

You may also choose to set `flow_names`, which enables indexing by name instead of by number:

```
class FibonacciNode(goos.ArrayFlowOpMixin, goos.ProblemGraphNode):

  def __init__(self, ...) -> None:
    super().__init__(...,
                     flow_types=[goos.Function, goos.Function],
                     flow_names=["first", "second"])
    ...

  ...

node = FibonacciNode(...)
out_sum = node["first"] * node["second"]
```

Using `IndexOp`

You can also "manually" extract an element from an `ArrayFlow` node by using the `IndexOp` node:

```
# Computes the next two elements in the Fibonacci sequence.
class FibonacciNode(goos.ProblemGraphNode):

  def __init__(self, in1: goos.Function, in2: goos.Function) -> None:
    super().__init__([in1, in2])
    ...

  def eval(self, inputs: List[goos.NumericFlow]) -> goos.ArrayFlow:
    fib_next = inputs[0].array + inputs[1].array
    fib_next_next = inputs[1].array + fib_next
    return goos.ArrayFlow([goos.NumericFlow(fib_next),
                           goos.NumericFlow(fib_next_next)])

  ...

node = FibonacciNode(...)
out_sum = (goos.cast(IndexOp(node, 0), goos.Function)
           * goos.cast(IndexOp(node, 1), goos.Function)
```

Note that we had to cast the output node into `goos.Function` before being able to use arithemetic operations. This arises from the fact that `IndexOp` inherits directly from `ProblemGraphNode`, so arithmetic operations, which can only operate on `Function` cannot be directly performed.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search