Band structure calculation of a photonic crystal slab

In this notebook, we simulate a photonic crystal slab consisting of a square lattice of air holes in a dielectric slab. Our goal is to compute the band structure of this photonic crystal slab, as found in:

> Shanhui Fan and J. D. Joannopoulos, "Analysis of guided resonances in photonic crystal slabs," Phys. Rev. B 65, 235112 (2002).

To this end, we excite the structure with several `PointDipole` sources, and we measure the response with several `FieldTimeMonitor` monitors. We excite modes with a fixed Bloch wavevector by using Bloch boundary conditions. We then use the `ResonanceFinder` to find the resonant frequencies. By sweeping the Bloch wavevector, we obtain the full band structure of the photonic crystal slab.

See also the api reference for `ResonanceFinder` here.

If you are new to the finite-difference time-domain (FDTD) method, we highly recommend going through our FDTD101 tutorials.

In [1]:
```
# standard python imports
import numpy as np
import matplotlib.pyplot as plt
import xarray as xr

import tidy3d as td
from tidy3d import web
from tidy3d.plugins.resonance import ResonanceFinder
```

We will randomly position our sources and monitors. Here, we seed the random number generator to guarantee reproducible results.

In [2]:
```
rng = np.random.default_rng(12345)
```

Now we define the parameters for the simulation, structure, sources, and monitors.

We take the dipole polarization to be `Hz` and the symmetry to be `(0,0,1)` in order to excite only modes which are even with respect to the xy mirror plane.

In [3]:
```
# Simulation parameters
runtime_fwidth = 200.0  # in units of 1/frequency bandwidth of the source
t_start_fwidth = 5.0  # time to start monitoring after source has decayed, un
dPML = 1.0  # space between PhC slabs and PML, in unit of longest wavelength

# Structure parameters (um)
a_lattice = 1  # lattice constant "a"
r_hole = 0.2 * a_lattice  # radius of the air holes
t_slab = 0.5 * a_lattice  # slab thickness
ep_slab = 12  # dielectric constant of the slab
ep_hole = 1  # dielectric constant of the holes
```

```
# Frequency range of interest (Hz)
freq_range_unitless = np.array((0.1, 0.43))  # in units of c/a
freq_scale = (
    td.constants.C_0 / a_lattice
)  # frequency scale determined by the lattice constant
freq_range = freq_range_unitless * freq_scale
lambda_range = (td.constants.C_0 / freq_range[1], td.constants.C_0 / freq_ran

# Gaussian pulse parameters
freq0 = np.sum(freq_range) / 2  # central frequency
freqw = 0.3 * (freq_range[1] - freq_range[0])  # pulse width

# Runtime
run_time = runtime_fwidth / freqw
print(f"Total runtime = {(run_time*1e12):.2f} ps")
t_start = t_start_fwidth / freqw

# Simulation size
spacing = dPML * lambda_range[-1]  # space between PhC slabs and PML
sim_size = Lx, Ly, Lz = (a_lattice, a_lattice, 2 * spacing + t_slab)

# Number of k values to sample, per edge of the irreducible Brillouin zone
Nk = 4

# Number of dipoles and monitors
num_dipoles = 7
num_monitors = 2

# Dipole polarization and symmetry
polarization = "Hz"
symmetry = (0, 0, 1)
```

```
Total runtime = 6.74 ps
```

We define the materials and structures in terms of the above parameters. We take the photonic crystal slab to be lying in the xy plane. Because the photonic crystal slab is periodic in the x and y directions, we only need to simulate a single unit cell, containing the dielectric slab and a single air hole.

In [4]:

```
mat_slab = td.Medium(permittivity=ep_slab, name="mat_slab")
mat_hole = td.Medium(permittivity=ep_hole, name="mat_hole")

slab = td.Structure(
    geometry=td.Box(
        center=(0, 0, 0),
        size=(td.inf, td.inf, t_slab),
    ),
    medium=mat_slab,
    name="slab",
)

hole = td.Structure(
    geometry=td.Cylinder(
        center=(0, 0, 0),
        axis=2,
        radius=r_hole,
        length=t_slab,
    ),
    medium=mat_hole,
    name="hole",
)

structures = [slab, hole]
```

We will excite the photonic crystal slab with several `PointDipole` sources. Each dipole will have a random position and phase.

In [5]:
```
dipole_positions = rng.uniform(
    [-Lx / 2, -Ly / 2, 0], [Lx / 2, Ly / 2, 0], [num_dipoles, 3]
)

dipole_phases = rng.uniform(0, 2 * np.pi, num_dipoles)

pulses = []
dipoles = []
for i in range(num_dipoles):
    pulse = td.GaussianPulse(freq0=freq0, fwidth=freqw, phase=dipole_phases[i
    pulses.append(pulse)
    dipoles.append(
        td.PointDipole(
            source_time=pulse,
            center=tuple(dipole_positions[i]),
            polarization=polarization,
            name="dipole_" + str(i),
        )
    )
```

We create `FieldTimeMonitors` to record the field as a function of time at several random locations within the photonic crystal slab. Crucially, we start the monitors after the source pulse has decayed.

In [6]:
```
monitor_positions = rng.uniform(
    [-Lx / 2, -Ly / 2, 0], [Lx / 2, Ly / 2, 0], [num_monitors, 3]
)

monitors_time = []
for i in range(num_monitors):
    monitors_time.append(
        td.FieldTimeMonitor(
            fields=["Hz"],
            center=tuple(monitor_positions[i]),
            size=(0, 0, 0),
            start=t_start,
            name="monitor_time_" + str(i),
        )
    )
```

We will perform `3*Nk` different simulations, each with different Bloch boundary conditions, as we sweep the Bloch wavevector over the boundary of the irreducible Brillouin zone. We sweep over three lines, namely ΓX, XM, and $M\Gamma$. We use a PML in the z direction.

Here, we simply define all of the boundary conditions we will use and put them into a single array.

In [7]:
```
bspecs_gammax = []
bspecs_xm = []
bspecs_mgamma = []
for i in range(Nk):
    bspecs_gammax.append(
        td.BoundarySpec(
            x=td.Boundary.bloch((1 / 2) * i / Nk),
            y=td.Boundary.periodic(),
```

```
            z=td.Boundary.pml(),
        )
    )
    bspecs_xm.append(
        td.BoundarySpec(
            x=td.Boundary.bloch(1 / 2),
            y=td.Boundary.bloch((1 / 2) * i / Nk),
            z=td.Boundary.pml(),
        )
    )
    bspecs_mgamma.append(
        td.BoundarySpec(
            x=td.Boundary.bloch((1 / 2) * (1 - i / Nk)),
            y=td.Boundary.bloch((1 / 2) * (1 - i / Nk)),
            z=td.Boundary.pml(),
        )
    )
bspecs = bspecs_gammax + bspecs_xm + bspecs_mgamma
```

Now we define the simulations we want to run.

In [8]:

```
sims = {}
for i in range(3 * Nk):
    sims[f"sim_{i}"] = td.Simulation(
        center=(0, 0, 0),
        size=sim_size,
        grid_spec=td.GridSpec.auto(),
        structures=structures,
        sources=dipoles,
        monitors=monitors_time,
        run_time=run_time,
        shutoff=0,
        boundary_spec=bspecs[i],
        normalize_index=None,
        symmetry=symmetry,
    )
```

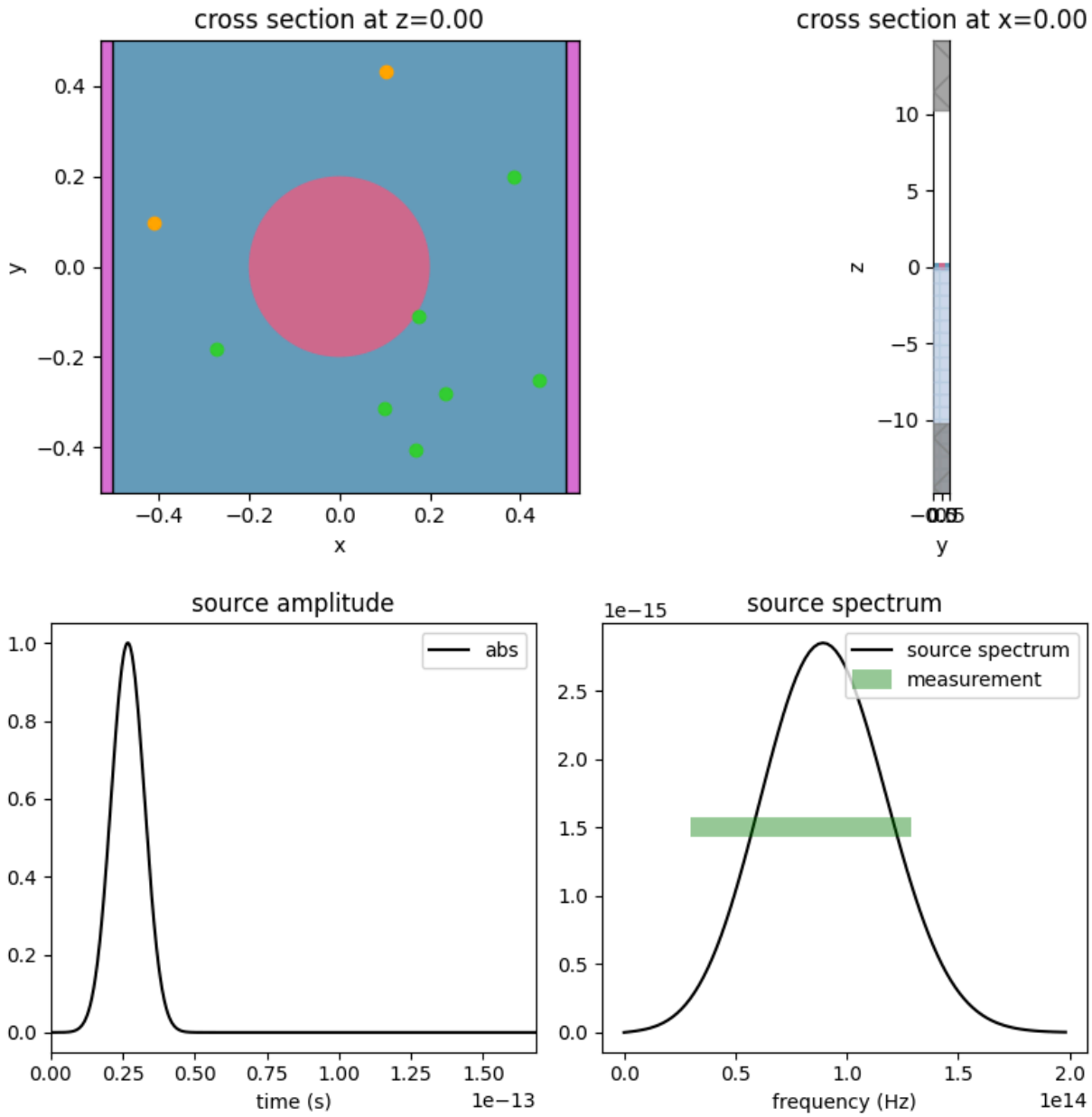
Let's check that the structure and source look correct. The source spectrum must fill the entire frequency range of interest.

In [9]:

```
fig, ax = plt.subplots(1, 2, tight_layout=True, figsize=(10, 4))
sims["sim_0"].plot(z=0.0, ax=ax[0])
sims["sim_0"].plot(x=0, freq=freq0, ax=ax[1])
plt.show()

f, (ax1, ax2) = plt.subplots(1, 2, tight_layout=True, figsize=(8, 4))
plot_time = 5 / freqw
ax1 = (
    sims["sim_0"]
    .sources[0]
    .source_time.plot(times=np.linspace(0, plot_time, 1001), val="abs", ax=ax
)
ax1.set_xlim(0, plot_time)
ax2 = (
    sims["sim_0"]
    .sources[0]
    .source_time.plot_spectrum(
        times=np.linspace(0, sims["sim_0"].run_time, 10001), val="abs", ax=ax
    )
)
ax2.hlines(1.5e-15, freq_range[0], freq_range[1], linewidth=10, color="g", al
```

```
ax2.legend(("source spectrum", "measurement"))
plt.show()
```



Now we run the simulations as a `Batch`.

We set `verbose=True` to keep track of the status of the jobs in the `Batch`.

In [10]:
```
# initialize a batch and run them all
batch = td.web.Batch(simulations=sims, verbose=True)

# run the batch and store all of the data in the `data/` dir.
batch_data = batch.run(path_dir="data")
```

```
10:32:45 -03 Created task 'sim_0' with task_id
             'fdve-c0a72a0d-28ff-4592-883d-50030bf85839' and task_type

             View task using web UI at
             'https://tidy3d.simulation.cloud/workbench?taskId=fdve-c0a
             f-4592-883d-50030bf85839'.
```

```
10:32:47 -03 Created task 'sim_1' with task_id
             'fdve-686c623f-3acd-4a21-90a8-35c049d842fe' and task_type

             View task using web UI at
             'https://tidy3d.simulation.cloud/workbench?taskId=fdve-68
             d-4a21-90a8-35c049d842fe'.

10:32:49 -03 Created task 'sim_2' with task_id
             'fdve-948fdcb0-9da5-4dc0-99ae-3ea1f5bdcc2f' and task_type

             View task using web UI at
             'https://tidy3d.simulation.cloud/workbench?taskId=fdve-948
             5-4dc0-99ae-3ea1f5bdcc2f'.

10:32:50 -03 Created task 'sim_3' with task_id
             'fdve-4db18d44-fbcf-479f-b9d3-4bba6aad61b4' and task_type

             View task using web UI at
             'https://tidy3d.simulation.cloud/workbench?taskId=fdve-4db
             f-479f-b9d3-4bba6aad61b4'.

10:32:52 -03 Created task 'sim_4' with task_id
             'fdve-f2352afa-b7a2-46c7-b6a0-18a6aeaa62bf' and task_type

             View task using web UI at
             'https://tidy3d.simulation.cloud/workbench?taskId=fdve-f23
             2-46c7-b6a0-18a6aeaa62bf'.

10:32:53 -03 Created task 'sim_5' with task_id
             'fdve-ac7c9002-6841-4a84-ac76-a179805c7c09' and task_type

             View task using web UI at
             'https://tidy3d.simulation.cloud/workbench?taskId=fdve-ac7
             1-4a84-ac76-a179805c7c09'.

10:32:55 -03 Created task 'sim_6' with task_id
             'fdve-3366543d-e1c2-4335-9944-d31b6fa369b0' and task_type
```

View task using web UI at
'https://tidy3d.simulation.cloud/workbench?taskId=fdve-336
2-4335-9944-d31b6fa369b0'.

10:32:56 -03 Created task 'sim_7' with task_id
'fdve-4b425659-89c6-4f0f-a9a8-5f82b141a547' and task_type

View task using web UI at
'https://tidy3d.simulation.cloud/workbench?taskId=fdve-4b4
6-4f0f-a9a8-5f82b141a547'.

10:32:58 -03 Created task 'sim_8' with task_id
'fdve-414b3dc7-e849-4f35-9b65-8f549365df22' and task_type

View task using web UI at
'https://tidy3d.simulation.cloud/workbench?taskId=fdve-414
9-4f35-9b65-8f549365df22'.

10:32:59 -03 Created task 'sim_9' with task_id
'fdve-48663d55-7ce3-4131-ae50-e8297750ff76' and task_type

View task using web UI at
'https://tidy3d.simulation.cloud/workbench?taskId=fdve-486
3-4131-ae50-e8297750ff76'.

10:33:01 -03 Created task 'sim_10' with task_id
'fdve-fb4ca11f-4322-4c7c-96cc-31cc6c0668d1' and task_type

View task using web UI at
'https://tidy3d.simulation.cloud/workbench?taskId=fdve-fb4
2-4c7c-96cc-31cc6c0668d1'.

10:33:02 -03 Created task 'sim_11' with task_id
'fdve-9ffca0b0-59bc-4ded-bd95-de6d9ff24499' and task_type

View task using web UI at
'https://tidy3d.simulation.cloud/workbench?taskId=fdve-9ff
c-4ded-bd95-de6d9ff24499'.

```
10:33:14 -03 Started working on Batch.
```

```
10:33:24 -03 Maximum FlexCredit cost: 0.300 for the whole batch.
             Use 'Batch.real_cost()' to get the billed FlexCredit cost
             Batch has completed.
```

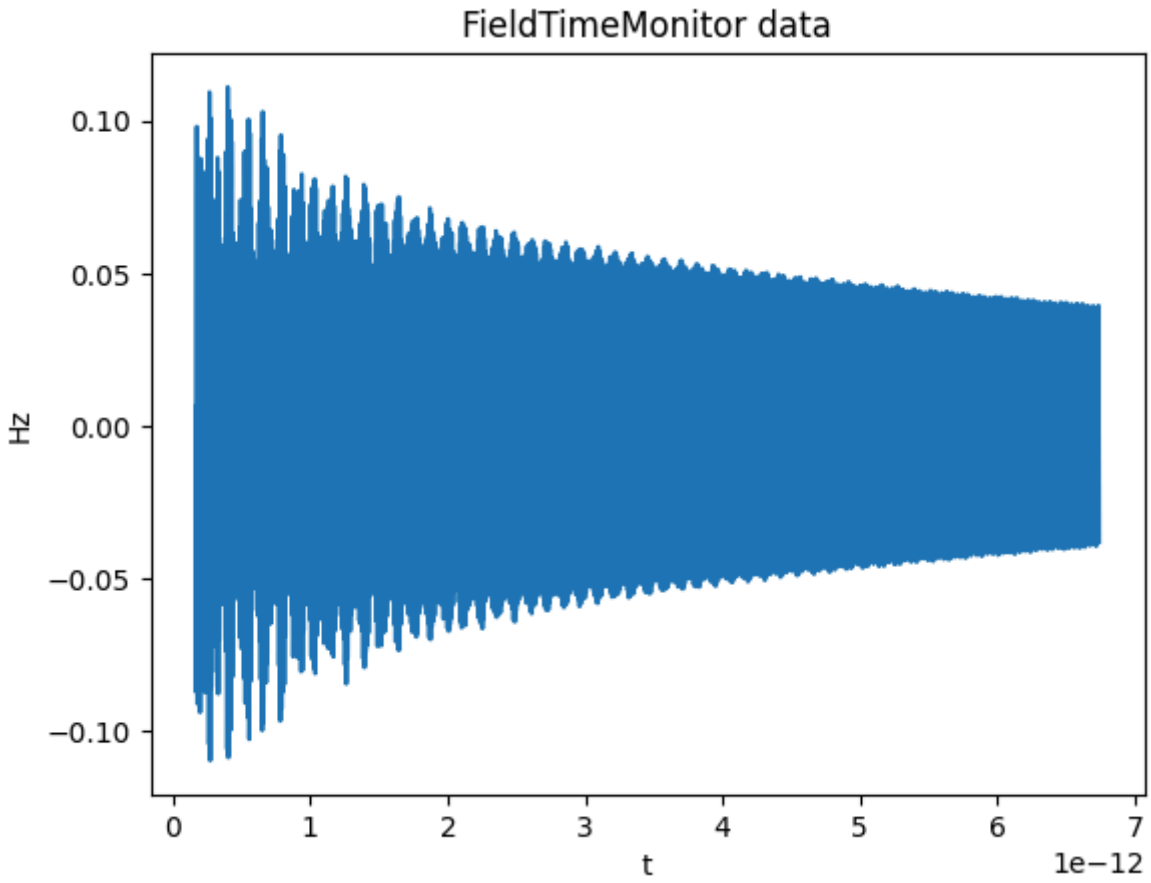
```
10:35:57 -03 Batch complete.
```

Now that the simulations are complete, we can analyze the data. Let's first look at one of the FieldTimeMonitors to make sure the source has decayed.

In [11]:
```
plt.plot(
    batch_data["sim_1"].monitor_data["monitor_time_0"].Hz.t,
    np.real(batch_data["sim_1"].monitor_data["monitor_time_0"].Hz.squeeze()),
)
plt.title("FieldTimeMonitor data")
plt.xlabel("t")
plt.ylabel("Hz")
plt.show()
```

```
10:38:08 -03 loading simulation from
             data/fdve-686c623f-3acd-4a21-90a8-35c049d842fe.hdf5
```

```
             loading simulation from
             data/fdve-686c623f-3acd-4a21-90a8-35c049d842fe.hdf5
```



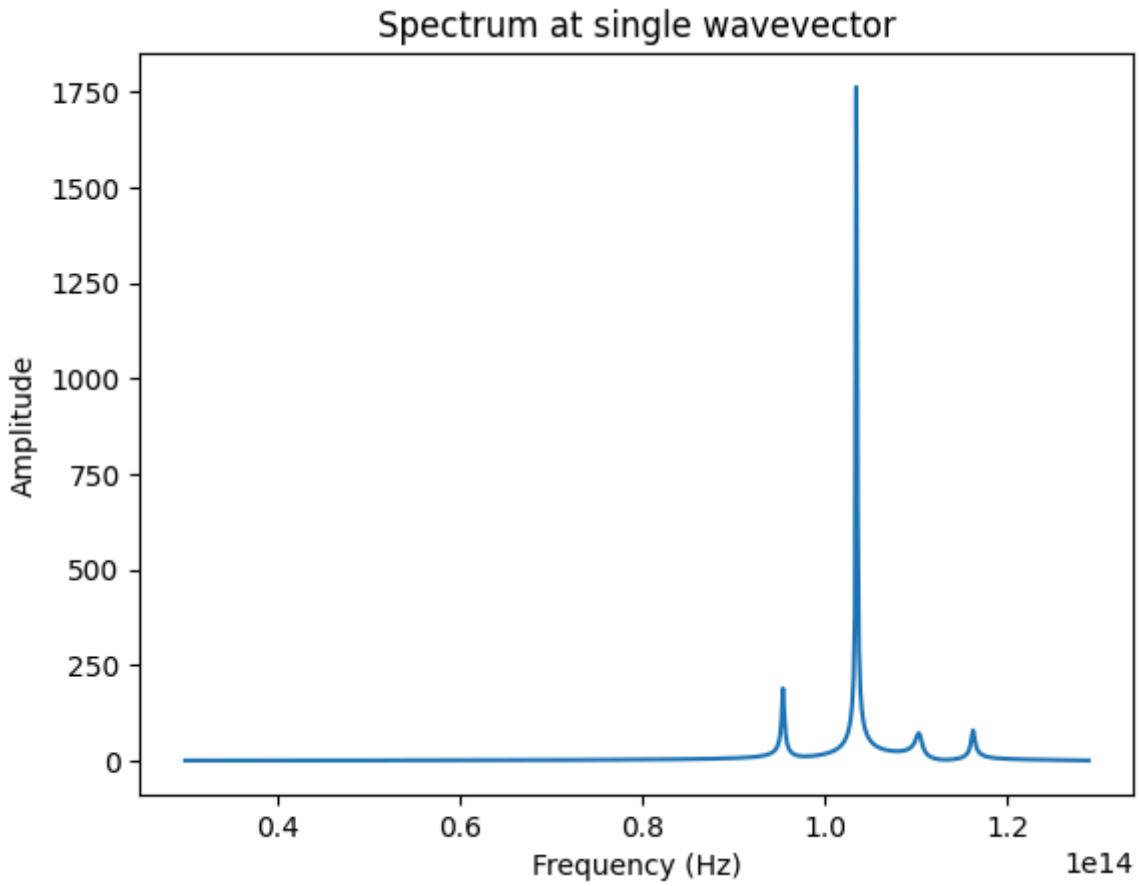
We see that the source has mostly decayed by the time we switch on the monitors, and the remaining data shows decay and oscillation due to the resonances inside the system.

Looking at the Fourier transform of this data, we can see resonances at the band frequencies.

In [12]:
```
df = 1 / np.amax(batch_data["sim_1"].monitor_data["monitor_time_0"].Hz.t)
minn = int(freq_range[0] / df)
maxn = int(freq_range[1] / df)
spectrum = np.fft.fft(batch_data["sim_1"].monitor_data["monitor_time_0"].Hz.s
plt.plot(
    np.linspace(freq_range[0], freq_range[1], maxn - minn),
    np.abs(spectrum[::-1][minn:maxn]),
)
plt.title("Spectrum at single wavevector")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude")
plt.show()
```

```
10:38:10 -03 loading simulation from
             data/fdve-686c623f-3acd-4a21-90a8-35c049d842fe.hdf5
```

```
             loading simulation from
             data/fdve-686c623f-3acd-4a21-90a8-35c049d842fe.hdf5
```



We use the ResonanceFinder plugin to find the band frequencies.

We first construct a `ResonanceFinder` object storing our parameters, and then call `run()` on our list of `FieldTimeData` objects. This will add up the signals from all of the monitors before searching for resonances. The `ResonanceFinder` class has additional methods in case the signal takes another form; see the api reference here.

The `run()` method returns an `xr.Dataset` containing the decay rate, Q factor, amplitude, phase, and estimation error for each resonance as a function of frequency.

In [13]:
```
resonance_finder = ResonanceFinder(freq_window=tuple(freq_range))
resonance_data = resonance_finder.run(signals=batch_data["sim_1"].data)
resonance_data.to_dataframe()
```

```
10:38:11 -03 loading simulation from
             data/fdve-686c623f-3acd-4a21-90a8-35c049d842fe.hdf5
```

Out[13]:

freq	decay	Q	amplitude	phase	error
2.527775e+13	2.296547e+10	3457.903341	0.001526	1.530347	0.000320
9.788251e+13	7.422102e+11	414.312534	0.009620	2.030329	0.000019
1.061235e+14	9.526460e+10	3499.692898	0.026843	-2.893037	0.000005
1.131264e+14	2.127493e+12	167.049748	0.035868	1.792082	0.000024
1.191562e+14	1.151470e+12	325.097664	0.020007	-2.087170	0.000019
1.372938e+14	1.121671e+12	384.534325	0.021763	-0.424613	0.000020

We see the four resonances from the previous figure. All four have reasonable Q factors, amplitudes, and errors, so they are likely to represent physical resonances. Note that in order to accurately obtain the Q factor for high-Q modes, it may be necessary to run the simulation for a longer time.

Now we are ready to compute the band structure. We run the resonance finder at every Bloch wavevector.

In [14]:
```
resonance_finder = ResonanceFinder(freq_window=tuple(freq_range))
resonance_datas = []
for i in range(3 * Nk):
    sim_data = batch_data[f"sim_{i}"]
    resonance_datas.append(resonance_finder.run(signals=sim_data.data))
```

```
10:38:14 -03 loading simulation from
             data/fdve-c0a72a0d-28ff-4592-883d-50030bf85839.hdf5
```



```
10:38:16 -03 loading simulation from
             data/fdve-686c623f-3acd-4a21-90a8-35c049d842fe.hdf5
```



```
10:38:22 -03 loading simulation from
             data/fdve-948fdcb0-9da5-4dc0-99ae-3ea1f5bdcc2f.hdf5
```



```
10:38:26 -03 loading simulation from
             data/fdve-4db18d44-fbcf-479f-b9d3-4bba6aad61b4.hdf5
```



```
10:38:30 -03 loading simulation from
             data/fdve-f2352afa-b7a2-46c7-b6a0-18a6aeaa62bf.hdf5
```



```
10:38:35 -03 loading simulation from
             data/fdve-ac7c9002-6841-4a84-ac76-a179805c7c09.hdf5
```



```
10:38:39 -03 loading simulation from
             data/fdve-3366543d-e1c2-4335-9944-d31b6fa369b0.hdf5
```



```
10:38:43 -03 loading simulation from
```

```
                data/fdve-4b425659-89c6-4f0f-a9a8-5f82b141a547.hdf5
```

```
10:38:47 -03 loading simulation from
             data/fdve-414b3dc7-e849-4f35-9b65-8f549365df22.hdf5
```



```
10:38:51 -03 loading simulation from
             data/fdve-48663d55-7ce3-4131-ae50-e8297750ff76.hdf5
```



```
10:38:55 -03 loading simulation from
             data/fdve-fb4ca11f-4322-4c7c-96cc-31cc6c0668d1.hdf5
```



```
10:39:00 -03 loading simulation from
             data/fdve-9ffca0b0-59bc-4ded-bd95-de6d9ff24499.hdf5
```



We define a function to filter resonances based on their Q, amplitude, and error.

In [15]:

```
def filter_resonances(resonance_data, minQ, minamp, maxerr):
    resonance_data = resonance_data.where(abs(resonance_data.Q) > minQ, drop=
    resonance_data = resonance_data.where(resonance_data.amplitude > minamp,
    resonance_data = resonance_data.where(resonance_data.error < maxerr, drop
    return resonance_data
```

We plot the band structure with the light line overlaid.

In [16]:

```
for i in range(3 * Nk):
    resonance_data = resonance_datas[i]
    resonance_data = filter_resonances(
        resonance_data=resonance_data, minQ=0, minamp=0.001, maxerr=100
    )
    freqs = resonance_data.freq.to_numpy()
    Qs = resonance_data.Q.to_numpy()
    plt.scatter(np.full(len(freqs), (1 / 2) * i / Nk), freqs / 3e14, color="b

lightx = np.linspace(0, 0.5, 100)
lighty1 = lightx
lighty3 = (0.5 - lightx) * np.sqrt(2)

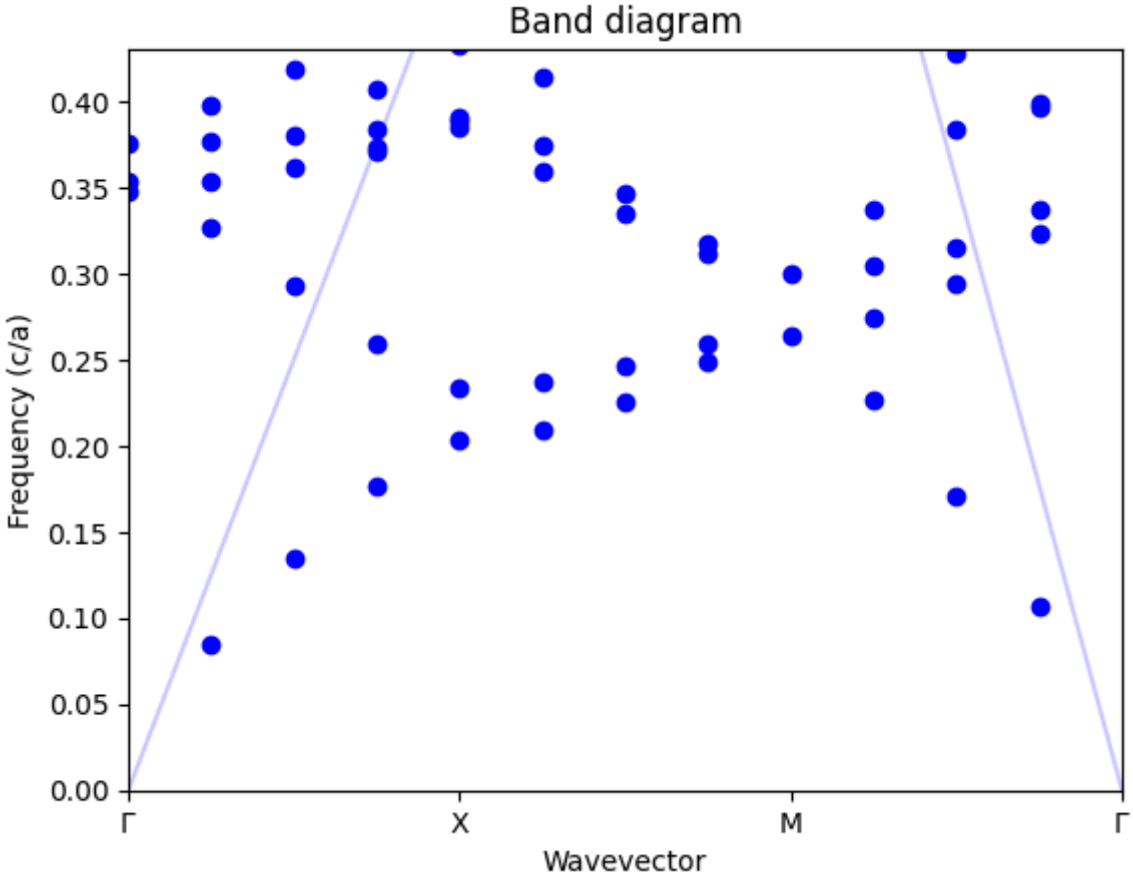
plt.plot(lightx, lighty1, color="blue", alpha=0.2)
plt.plot(1 + lightx, lighty3, color="blue", alpha=0.2)

plt.ylim(0, freq_range_unitless[1])

plt.title("Band diagram")
plt.ylabel("Frequency (c/a)")
plt.xlabel("Wavevector")
plt.xticks([0, 0.5, 1, 1.5], ["$\Gamma$", "X", "M", "$\Gamma$"])
```

```
plt.xlim(0, 1.5)
plt.show()
```

```
<>:22: SyntaxWarning: invalid escape sequence '\G'
<>:22: SyntaxWarning: invalid escape sequence '\G'
<>:22: SyntaxWarning: invalid escape sequence '\G'
<>:22: SyntaxWarning: invalid escape sequence '\G'
/tmp/ipykernel_27549/2520820917.py:22: SyntaxWarning: invalid escape sequence
'\G'
  plt.xticks([0, 0.5, 1, 1.5], ["$\Gamma$", "X", "M", "$\Gamma$"])
/tmp/ipykernel_27549/2520820917.py:22: SyntaxWarning: invalid escape sequence
'\G'
  plt.xticks([0, 0.5, 1, 1.5], ["$\Gamma$", "X", "M", "$\Gamma$"])
```



The bandstructure we obtained matches the expected result from the paper. If we were seeing too many resonances, we could change the parameters to our filter_resonances function to eliminate the spurious ones. If we were seeing too few resonances even before filtering, we might have to change the parameters of the ResonanceFinder, for example decreasing `rcond` or increasing `init_num_freqs`. If the ResonanceFinder takes too long, we can decrease `init_num_freqs`. There can also be resonances on the light line associated with Wood's anomaly; we filter those out here based on their small amplitude.