
# NumBAT - The Numerical Brillouin Analysis Tool

***Release 2.1.4***

**Michael Steel, Bjorn Sturmberg, Blair Morrison, Mike Smith and Christopher Poulton**

**03 August 2025**

# CONTENTS:

Release date: 03 August 2025

CHAPTER

ONE

# INTRODUCTION TO NUMBAT


## 1.1 Introduction

NumBAT, the *Numerical Brillouin Analysis Tool*, is a software tool integrating electromagnetic and acoustic mode solvers to calculate the interactions of optical and acoustic waves in waveguides. Most notably, this includes Stimulated Brillouin Scattering (SBS) frequency shifts and optical gains.

This chapter provides some background on the capabilities and techniques used in NumBAT. If you would like to get straight to computations, jump ahead to the installation and setup instructions in *Installing NumBAT*.

## 1.2 Goals

NumBAT is designed primarily to calculate the optical gain response from stimulated Brillouin scattering (SBS) in integrated waveguides. It uses finite element algorithms to solve the electromagnetic and acoustic modes of a wide range of 2D waveguide structures. It can account for photoelastic/electrostriction and moving boundary/radiation pressure effects, as well as arbitrary acoustic anisotropy.

NumBAT also supports user-defined material properties and we hope its creation will drive a community-driven set of standard properties and geometries which will allow all groups to test and validate each other's work.

A full description of NumBAT's physical and numerical algorithms is available in the article B.C.P Sturmberg at al., "Finite element analysis of stimulated Brillouin scattering in integrated photonic waveguides", *J. Lightwave Technol.* **37**, 3791-3804 (2019), available at https://dx.doi.org/10.1109/JLT.2019.2920844.

NumBAT is open-source software and the authors welcome additions to the code. Details for how to contribute are available in *Contributing to NumBAT*.

## 1.3 Development team

NumBAT was originally developed by Bjorn Sturmberg, Kokou Dossou, Blair Morrison, Chris Poulton and Michael Steel in a collaboration between Macquarie University, the University of Technology Sydney, and the University of Sydney. Continuing development is by Michael Steel at Macquarie University.

We thank Christian Wolff, Mike Smith and Mikolaj Schmidt for contributions.

## 1.4 Citing NumBAT

If you use NumBAT in published work, we would appreciate a citation to B.C.P Sturmberg at al., "Finite element analysis of stimulated Brillouin scattering in integrated photonic waveguides", *J. Lightwave Technol.* **37**, 3791-3804 (2019), available at https://dx.doi.org/10.1109/JLT.2019.2920844 and https://arxiv.org/abs/1811.10219, and a link to the github page at https://github.com/michaeljsteel/NumBAT.

## 1.5 Contributing to NumBAT

NumBAT is open source software licensed under the GPL with all source and documentation available at github.com. We welcome additions to NumBAT code, documentation and the materials library. Interested users should fork the standard release from github and make a pull request when ready. For major changes, we strongly suggest contacting the NumBAT team before starting work at `michael.steel@mq.edu.au`.

## 1.6 Seeking assistance

We will do our best to support users of NumBAT within the time available. All requests should be sent to `michael.steel@mq.edu.au`.

- **For assistance with installing and building NumBAT, please see the instructions**
  in *Seeking help with building NumBAT* and collect all the required information before writing.
- For assistance with calculations once NumBAT is working, please send an email containing the following information:
  - Your platform (Linux, Windows, MacOS) and specific operating system
  - **The version number of your NumBAT installation.**
    You can determine this by typing the following command in your terminal

    ```
    python -c 'import numbat;print(numbat.version())'
    ```

  - **A short python script which demonstrates the issue you are trying to solve.**
    Please make the script as short in both code length and execution time as possible while still exhibiting the issue of concern.

**Note**: NumBAT is updated frequently with new features, API and bug fixes. Consequently, please understand that we can only provide assistance for issues with the *current* github version of NumBAT. Please be sure that your script runs against the current github tree and that your problem is still present with that version.

## 1.7 Release notes

### 1.7.1 Version 2.0

A number of API changes have been made in NumBAT 2.0 to tidy up the interface and make plotting and analysis simpler and more powerful. You will need to make some changes to existing files to run in NumBAT 2.0. Your best guide to new capabilities and API changes is to look through the code in the tutorial examples.

Some key changes you will need to make are as follows:

- On Linux, the fortran Makefile is now designed to work with a virtual environment python to avoid dependencies on your system python.
- There is a new core NumBAT module `numbat` that should be imported before any other NumBAT modules.
- It should no longer be necessary to import the `object` or `Numbat` (note different case) modules.
- The first call to any NumBAT code should be to create a NumBAT application object by calling `nbapp = numbat.NumBATApp()`.

- The default output prefix can now be set as an argument to `numbat.NumBATApp()`. All output can be directed to a sub-folder of the starting directory with a second argument: `nbapp = numbat.NumBATApp('tmp', 'tmpdir')`.
- The waveguide class `Struct` has been renamed to `Structure`.
- A waveguide is now constructed using `nbapp.make_waveguide` rather than `object.Structure`.
- The parameter names for some waveguide types have changed to become more intuitive than the generic `inc_a_x` approach. You can find details for a given structure using `NumBATApp(). _wg_structure_help(inc_shape)`.
- The interface for creating materials has changed. You now call the `materials.make_material(` *name* `)` function. For example `material_a = materials.make_material('Vacuum')`.
- To access an existing material in an existing `Structure` object (say, in a variable called `wguide`) use `wguide.get_material(` *label* `)` For example, `mat_a = wguide.get_material('b')` where the allowed labels are `bkg` and the letters `a` to `r`.
- The member name for refractive index in a `Material` object has changed from `n` to `refindex_n`.
- The member name for density in a `Material` object has changed from `n` to `rho`.
- Due to a change in parameters, the function `plotting.gain_spectra` is deprecated and replaced by `plotting.plot_gain_spectra` with the following changes:
  - The frequency arguments `freq_min` and `freq_max` should now be passed in units of Hz, not GHz.
  - The argument `k_AC` has been removed.
- In all functions the parameter `prefix_str` has been renamed to `prefix` for brevity. Using the default output settings in `NumBATApp()`, these should be rarely needed.
- All waveguides are now specified as individual plugin classes in the files `backend/msh/user_waveguides.json` and `backend/msh/user_meshes.py`. These files provide useful examples of how to design and load new waveguide templates. See the following chapter for more details.

## 1.8 About our mascot

The **numbat** (*Myrmecobius fasciatus*) is a delightful insect-eating marsupial from Western Australia, of which it is the official state animal. It has two other common Aboriginal names, *noombat* in the Nyungar language, and *walpurti* in the Pitjantjatjara language.

As a carnivorous marsiupial, they belong to the order Dasyuromorphia, closely related to quolls and the famed thylacines which had similar markings on their lower back. Once found across southern Australia, numbats are now confined to small local groups in Western Australia and the species has Endangered status.

Fig. 1: A numbat at Perth zoo in 2010. (Creative commons).

Apart from the distinctive striped back (which we like to think of as an acoustic wave made flesh), numbats have a number of unique properties. They are the only fully diurnal marsupial. They are insectivores and eat exclusively termites, perhaps 20000 each day!

To find out how you can support the care and revitalisation of this beautiful animal, check out the work at project-numbat and the Australian Wildlife Conservancy.

## 1.9 Acknowledgements

We acknowledge the elders past and present of the following First Nations people, on whose unceded lands NumBAT has been developed: the Wallumattagal clan of the Dharug nation, the Cammeraygal people, and the Gadigal people of the Eora nation.

Development of NumBAT has been supported in part by the Australian Research Council under Discovery Projects DP130100832, DP160101691, DP200101893 and DP220100488.

CHAPTER

TWO

# BACKGROUND THEORY

## 2.1 What does NumBAT actually calculate?

NumBAT performs three main types of calculations given a particular waveguide design:

- solve the electromagnetic modal problem using the finite element method (FEM).
- solve the elastic modal problem using FEM.
- calculate Brillouin gain coefficients and linewidths for a given triplet of two optical and one elastic mode, and use this to generate gain spectra.

Here we specify the precise mathematical problems being solved. For further details, see the NumBAT paper [1] in the Journal of Lightwave Technology at https://dx.doi.org/10.1109/JLT.2019.2920844.

### 2.1.1 Electromagnetic modal problem

The electromagnetic wave problem is defined by the vector wave equation

$$-\nabla \times (\nabla \times \vec{E}) + \omega^2 \epsilon_0 \, \epsilon_r(x,y) \vec{E} = 0,$$

where $\omega = 2\pi/\lambda$ is the optical angular frequency with $\lambda$ the free space wavelength. The waveguide properties are defined by the spatially varying relative dielectric constant $\epsilon_r(x,y)$.

The *real-valued* electric field $\vec{E}(x,y,z,t)$ has the following form for modal propagation along $z$:

$$\begin{aligned} \vec{E}(x,y,z,t) &= \left( \vec{\mathcal{E}}(\vec{r}) e^{-i\omega t} + \vec{\mathcal{E}}^*(\vec{r}) e^{i\omega t} \right) \\ &= \left( a\vec{e}(x,y) e^{i(kz-\omega t)} + a^* \vec{e}^*(x,y) e^{-i(kz-\omega t)} \right), \end{aligned}$$

in terms of the wavenumber $k$, complex field amplitude $\vec{\mathcal{E}}(\vec{r})$, mode profile $\vec{e}(x,y)$ and the mode amplitude $a$. Note that some authors include a factor of $\frac{1}{2}$ in these definitions which leads to slightly different expressions for the energy and power flow below.

By Faraday's law the complex magnetic field amplitude is given by

$$\vec{\mathcal{B}} = \frac{1}{i\omega} \nabla \times \vec{\mathcal{E}}.$$

#### Dependent and independent variables in the modal dispersion

In NumBAT, the electromagnetic mode problem is formulated with the optical angular frequency $\omega$ as the independent variable and the wavenumber or *propagation constant* $k(\omega)$ as the eigenvalue. For a given frequency, NumBAT thus finds the eigenvalues $k_n(\omega)$ and eigenmodes $\vec{e}_n(x,y)$. The algorithm finds the most-bound or "lowest" eigenvalues first. Thus the values $k_n(\omega)$ *decrease* with $n$ and there are only a finite number of guided mode solutions for a given structure and frequency $\omega$. Note that the propagation constant is also often denoted by $\beta_n(\omega)$.

It is thus straightforward to generate dispersion relations $k_n(\omega)$ on an equal-spaced frequency grid. If you require the wavenumber on an equal-spaced grid, you will need to interpolate the $k_n(\omega)$ functions.

### 2.1.2 Elastic modal problem

The elastic modal problem is defined by the wave equation

$$\nabla \cdot \bar{T} + \Omega^2 \rho(x,y)\vec{U} = 0,$$

where $\Omega$ is the elastic angular frequency and $\vec{U}(x,y,z)$ is the elastic displacement field. Further, $\bar{T} = \mathbf{c}(x,y)\bar{S}$ is the rank-2 stress tensor, defined in terms of the rank-4 stiffness tensor $\mathbf{c}$ and the rank-2 strain tensor $\bar{S} = S_{ij} = \frac{1}{2}(\frac{\partial U_i}{\partial r_j} + \frac{\partial U_j}{\partial r_i})$.

The waveguide properties are defined by the spatially varying density and stiffness functions.

The displacement field has the modal propagation form

$$\vec{U} = b(z)\,\vec{u}(x,y)e^{i(qz-\Omega t)} + b^*(z)\,\vec{u}(x,y)^* e^{-i(qz-\Omega t)}.$$

where $q$ is the elastic wavenumber or propagation constant.

#### Dependent and independent variables in the modal dispersion

In NumBAT, the elastic mode problem is formulated in the *opposite sense* to the electromagnetic problem. In this case, the elastic wavenumber $q$ is the independent variable, and NumBAT solves for the corresponding angular frequency eigenvalues $\Omega_n(q)$ and eigenmodes $\vec{u}_n(x,y)$. With the wavenumber as the independent variable, the eigenvalues $q_n(\Omega)$ *increase* with the mode index $n$. In such a formulation, there is no upper bound to the number of physical eigenstates, other than the numerical resolution of the FEM grid. In practice, above frequencies of $\Omega/(2\pi)$ of a few tens of GHz, the elastic losses become so large that the propagation is restricted to distances of only a few microns.

It is thus straightforward to generate dispersion relations $\Omega_n(q)$ on an equal-spaced wavenumber grid. If you require the frequency on an equal-spaced grid, you will need to interpolate the $\Omega_n(q)$ functions.

For details on how these problems are framed as finite element problems, we refer to Ref. [1], though some details are provided in *Numerical formulation of the finite element problems*.

### 2.1.3 Mode normalisation

To calculate nonlinear quantities correctly, it is necessary to account for the energy flux in each mode. This can be done either by scaling the field amplitudes so that they carry a fixed energy flux (say 1 W) and accounting for this in the nonlinear coefficients, or by allowing an arbitrary field normalisation and explicitly including the power flux into the gain calculation. The code in NumBAT follows this second approach which has the advantage that the numerical fields can be scaled to suit other purposes (for instance convenient plotting scales) without affecting the nonlinear coefficient calculations.

For propagation in a given mode $\vec{e}_n$ or $\vec{U}_n$, the optical ($o$) and elastic ($a$ for *acoustic*) energy fluxes in Watts, and linear energy densities in (J/m), are given by the following expressions

$$\mathcal{P}_n^{(o)} = 2\mathrm{Re}\int \mathrm{d}^2 r\, \hat{z}\cdot(\vec{e}_n^*(x,y)\times\vec{h}_n(x,y)),$$

$$\mathcal{E}_n^{(o)} = 2\epsilon_0 \int \mathrm{d}^2 r\, \epsilon_r(x,y)|\vec{e}_n(x,y)|^2$$

$$\mathcal{P}_n^{(a)} = \mathrm{Re}\int \mathrm{d}^2 r\, (-2i\Omega)\sum_{jkl} c_{zjkl}(x,y)u_{mj}^*(x,y)\partial_k u_{ml}(x,y)$$

$$\mathcal{E}_n^{(a)} = 2\Omega^2 \int_A \mathrm{d}^2 r\, \rho(x,y)|\vec{u}_n(x,y)|^2.$$

For fields with slowly-varying optical and elastic amplitudes $a_m(z)$ and $b_m(z)$, the total carried powers are

$$P^{(o)}(z) = \sum_n |a_n(z)|^2 \mathcal{P}_n^{(o)}$$

$$P^{(a)}(z) = \sum_n |b_n(z)|^2 \mathcal{P}_n^{(a)}.$$

Note that in this convention, the amplitude functions $a_n(z)$ and $b_n(z)$ are dimensionless and the dimensionality of the fields lives in the modal functions $\vec{e}_n, \vec{h}_n, \vec{U}_n$, which are taken to have their conventional SI units.

### 2.1.4 SBS gain calculation modal problem

The photoelastic and moving boundary couplings in J/m are given by

$$Q^{(\mathrm{PE})} = -\epsilon \int_A \mathrm{d}^2 r \sum_{ijkl} \epsilon_r^2 \, e_i^{(s)*} \, e_j^{(p)} \, p_{ijkl} \, \partial_k u_l^*$$

$$Q^{(\mathrm{MB})} = \int_{\mathcal{C}} \mathrm{d}\vec{r} \, (\vec{u}^* \cdot \hat{n}) \times \left[ (\epsilon_a - \epsilon_b)\epsilon_0 (\hat{n} \times \vec{u}^{(s)})^* \cdot (\hat{n} \times \vec{e}^{(p)}) - (\epsilon_a^{-1} - \epsilon_b^{-1})\epsilon_0^{-1} (\hat{n} \cdot \vec{d}^{(s)})^* \cdot (\hat{n} \cdot \vec{d}^{(p)}) \right]$$

Note that in general these functions are complex, rather than purely real or imaginary. The equations of motion in the next section show how this is consistent with energy conservation requirements.

Then, at least for backward SBS, the peak SBS gain of the Stokes wave $\Gamma$ is given by

$$\Gamma = \frac{2\omega\Omega}{\alpha_t} \frac{|Q_{\mathrm{tot}}|^2}{\mathcal{P}^{(s)}\mathcal{P}^{(p)}\mathcal{E}^{(a)}},$$

where the total SBS coupling is $Q_{\mathrm{tot}} = Q^{(\mathrm{PE})} + Q^{(\mathrm{MB})}$. As anticipated above, the modal power fluxes appear in the denominator of this expression.

The quantity $\alpha_t$ is the *temporal* elastic loss coefficient in $\mathrm{s}^{-1}$. It is related to the spatial attenuation coefficient by $\alpha_s = \alpha_t / v_{\mathrm{p}}^{(\mathrm{a})}$ with $v_{\mathrm{p}}^{(\mathrm{a})}$ being the elastic phase velocity.

In a backward SBS problem, where there is genuine gain in Stokes optical field propagating in the negative $z$ direction, its optical power evolves as

$$P^{(\mathrm{s})}(z) = P_{\mathrm{in}}^{(\mathrm{s})} e^{-\Gamma z}.$$

In forward Brillouin scattering, the same equations for the couplings $Q_i$ apply, but it is generally more helpful to think in terms of the spectral processing of the Stokes field rather than a simple gain. For this reason, we prefer the general term "forward Brillouin scattering" to "forward SBS", which you may also encounter. See refs. [2] [3] for detailed discussion of this issue.

### 2.1.5 SBS equations of motion

With the above conventions, the dynamical equations for the slowly-varying amplitudes are

$$\frac{1}{v_g^p}\frac{\partial}{\partial t} a_p + \frac{\partial}{\partial z} a_p = -i \frac{\omega_p Q_{\mathrm{tot}}}{\mathcal{P}_o} a_s b$$

$$\frac{1}{v_g^s}\frac{\partial}{\partial t} a_s - \frac{\partial}{\partial z} a_s = i \frac{\omega_s Q_{\mathrm{tot}}^*}{\mathcal{P}_o} a_p b^*$$

$$\frac{1}{v_g^a}\frac{\partial}{\partial t} b + \frac{\partial}{\partial z} b + \frac{\alpha_s}{2} b = -i \frac{\Omega Q_{\mathrm{tot}}^*}{\mathcal{P}_a} a_p a_s^*$$

Here we've chosen the group velocities to be positive and included the propagation direction explicitly. Note that the coupling $Q_{\mathrm{tot}}$ enters both in native and complex conjugate form. This ensures the total energy behaves appropriately.

### 2.1.6 Connecting these to output quantities from code

### 2.1.7 Equivalent forms of equations

- TODO: show forms without the normalisation energies and with cubic style effective area.
- Compare to some fiber literature and the hydrodynamic reprn.
- Connect to optical forces picture

## 2.2 The method of solution in NumBAT

NumBAT solves both the electromagnetic and elastic modal properties using *finite element method* (FEM) algorithms. Chapter *Numerical formulation of the finite element problems* provides a summary of the finite element formulation.

For further details, see refs. [1], [4] and [5].

## 2.3 Suggested reading on SBS and opto-elastic interactions in nanophotonics

A very extensive literature on SBS and related effects in nanophotonics has arisen over the period since 2010. Here we provide a few suggestions for entering and navigating that literature.

### 2.3.1 Books

The centenary of Brillouin scattering was marked with the publication of a two-volume book featuring contributions from many of the leading researchers in the field. These books provide detailed background and the history, theory, observation and application of Brillouin scattering in guided wave systems.

1. *Brillouin Scattering, Parts 1 and 2*, eds: B.J. Eggleton, M.J. Steel, C.G. Poulton, (Academic, 2022). https://doi.org/10.1016/S0080-8784(22)00024-2

### 2.3.2 Reviews

There are several excellent reviews covering the theory and experiment of Brillouin photonics.

1. A. Kobyakov, M. Sauer, and D. Chowdhury, "Stimulated Brillouin scattering in optical fibers," *Adv. Opt. Photon.* **2**, 1-59 (2010), https://doi.org/10.1364/AOP.2.000001.
2. B.J. Eggleton, C.G. Poulton, P.T. Rakich, M.J. Steel and G. P. Bahl, "Brillouin integrated photonics," *Nat. Photonics* **13**, 664–677 (2019), https://doi.org/10.1038/s41566-019-0498-z.
3. G.S. Wiederhecker, P. Dainese, T.P. Mayer Alegre, "Brillouin optomechanics in nanophotonic structures," *APL Photonics* **4**, 071101 (2019), https://doi.org/10.1063/1.5088169.

### 2.3.3 Theoretical development

The following papers feature more depth on the theory of SBS in waveguides. Chapters 2 and 3 of the *Brillouin Scattering* book listed above are also thorough resources for this material.

1. P.T. Rakich, C. Reinke, R. Camacho, P. Davids, and Z. Wang, "Giant Enhancement of Stimulated Brillouin Scattering in the Subwavelength Limit," *Phys. Rev. X* **2**, 011008 (2012). https://doi.org/10.1103/PhysRevX.2.011008
2. C. Wolff, M.J. Steel, B.J. Eggleton, and C.G. Poulton "Stimulated Brillouin scattering in integrated photonic waveguides: Forces, scattering mechanisms, and coupled-mode analysis," *Phys. Rev. A* **92**, 013836 (2015). https://doi.org/10.1103/PhysRevA.92.013836
3. J.E. Sipe and M.J. Steel, "A Hamiltonian treatment of stimulated Brillouin scattering in nanoscale integrated waveguides," *New J. Phys.* **18**, 045004 (2016). https://doi.org/10.1088/1367-2630/18/4/045004
4. B.C.P Sturmberg et al., "Finite element analysis of stimulated Brillouin scattering in integrated photonic waveguides", *J. Lightwave Technol.* **37**, 3791-3804 (2019). https://dx.doi.org/10.1109/JLT.2019.2920844
5. C. Wolff, M.J.A. Smith, B. Stiller, and C.G. Poulton, "Brillouin scattering—theory and experiment: tutorial," *J. Opt. Soc. Am. B* **38**, 1243-1269 (2021). https://doi.org/10.1364/JOSAB.416747

CHAPTER

THREE

# INSTALLING NUMBAT

This chapter provides instructions on installing NumBAT on each platform. Please email michael.steel@mq.edu.au to let us know of any difficulties you encounter, or suggestions for improvements to the install procedure on any platform, but especially for MacOS or Windows.

While NumBAT is developed on Linux, it can also be built natively on both MacOS and Windows. A pre-built executable version is also available for Windows, though it is updated less frequently than the main source code tree. The Linux builds can also be run under virtual machines on MacOS and Windows if desired.

In all cases, the current source code for NumBAT is hosted here on Github. Please always download the latest release from the github page.

You can now skip forward to the section for your operating system: *Linux*, *MacOS* or *Windows*.

**Installation instructions for your OS**

## 3.1 Installing on Linux

### 3.1.1 Install locations

There is no need to install NumBAT in a central location such as `/usr/local/` or `/opt/local/` though you may certainly choose to do so.

Here and throughout this documentation, we use the string `<NumBAT>` to indicate the root NumBAT install directory (e.g. `/usr/local/NumBAT`, `/home/mike/NumBAT`, `/home/myuserid/research/NumBAT`).

### 3.1.2 Requirements

NumBAT is developed and tested using relatively recent operating system, compiler and libraries. You should not need the very latest releases, but in general compilation will be smoother on an up-to-date system. In particular, we recommend using:

- an OS release from 2023 or later (eg Ubuntu 24.04/24.10)
- gcc compiler of version 13.0 or later
- python 3.11 or later

NumBAT is currently developed and tested on Ubuntu 25.04 with the following package versions: Python 3.13, Numpy 2.0, Arpack-NG, Suitesparse 7.1.0, and Gmsh 4.8.4. NumBAT also depends on the BLAS linear algebra library. We strongly recommend linking NumBAT against an optimised version, such as the MKL library provided in the free Intel OneAPI library (for Intel CPUs) or the AMD Optimizing CPU Libraries (AOCL) for AMD CPUs. The steps below demonstrate the Intel OneAPI approach.

NumBAT has also been successfully installed by users on Debian and RedHat/Fedora, and with different versions of packages, but these installations have not been as thoroughly documented so may require user testing. In general, any relatively current Linux system should work without trouble.

NumBAT building and installation is easiest if you have root access, but it is not required. See the section below if you do not have root access (or the ability to run `sudo`) on your machine.

The following steps use package syntax for Ubuntu/Debian systems. For other Linux flavours, you may need to use different package manager syntax and/or slightly different package names.

### 3.1.3 Required libraries

1. Before installing, ensure your system is up to date

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

2. Install the required libraries using your distribution's package manager.

   On Ubuntu, perform the following

```
$ sudo apt-get install gcc gfortran make gmsh python3-venv python3-dev meson pkg-
↪config ninja-build

$ sudo add-apt-repository universe

$ sudo apt-get install libarpack2-dev libparpack2-dev libatlas-base-dev libblas-
↪dev liblapack-dev  libsuitesparse-dev
```

3. If you wish to use the Intel OneAPI math libraries, you need both of the following:

   - ***Intel OneAPI Base Toolkit*:**
     This is the main Intel developer environment including C/C++ compiler and many high performance math libraries.

     Download and run the installer accepting all defaults.

   - ***Intel OneAPI HPC Toolkit***
     This adds the Intel Fortran compiler amongst other HPC tools.

     Download and run the installer accepting all defaults.

4. If you using the Intel OneAPI math libraries, you should add the library path `/opt/intel/oneapi/<release>/lib` to your `LD_LIBRARY_PATH` variable in one of your shell startup files (eg. `~/.bashrc`). Replace `<release>` with the correct string `2024.1` or similar depending on your installed version of OneAPI.

### 3.1.4 Building NumBAT itself

1. Create a python virtual environment for working with NumBAT. You can use any name and location for your tree. To specify a virtual environment tree called *nbpy3* in your home directory, enter

```
$ cd ~
$ python3 -m venv nbpy3
```

2. Activate the new python virtual environment

```
$ source ~/nbpy3/bin/activate
```

3. Install necessary python libraries

```
$ pip3 install numpy matplotlib scipy psutils
```

4. Create a working directory for your NumBAT work and move into it. From now, we will refer to this location as `<NumBAT>`.

5. To download the current version from the git repository and install any missing library dependencies, use

```
$ git clone https://github.com/michaeljsteel/NumBAT.git
$ cd NumBAT
```

6. Move to the `backend\fortran` directory.
    1. To build with the `gcc` compilers, run:

    ```
    $ make gcc
    ```

    2. To build with the Intel compilers, edit the file `nb-linuxintel-native-file.ini` adjusting the variables to point the correct location of the Intel compilers. Then run:

    ```
    $ make intel
    ```

7. If all is well, this will run to completion. If you encounter errors, please check that all the instructions above have been followed accurately. If you are still stuck, see *Troubleshooting Linux installs* for further ideas.
8. If you hit a compile error you can't resolve, please see the instructions at *Seeking help with building NumBAT* on how to seek help.
9. Once the build has apparently succeeded, it is time to test the installation with a short script that tests whether required applications and libraries can be found and loaded. Perform the following commands:

    ```
    $ cd <NumBAT>/backend
    $ python ./nb_install_tester.py
    ```

10. If this program runs without error, congratulations! You are now ready to proceed to the next chapter to begin using NumBAT
11. Once again, if you run into trouble, please don't hesitate to get in touch for help using the instructions at *Seeking help with building NumBAT*. Please do send all the requested information, as it usually allows us to solve your problem faster.

### 3.1.5 Using the Intel Fortran compiler

The default compiler for Linux is GCC's `gfortran`.

It is also possible to build NumBAT with the `ifx` compiler from Intel's free OneAPI HPC toolkit. You may find some modest performance improvements.

To use the Intel compiler,

1. If you have not already done so, install the Intel OneAPI Base and HPC Toolkits as described above.
2. Adjust your LD_LIBRARY_PATH variable in your `~/.bashrc` or equivalent to include `/opt/intel/oneapi/<release>/lib`. (Replace `<release>` with the correct string `2024.1` or similar depending on your installed version of OneAPI.)
3. In `<NumBAT>/backend/fortran`, repeat all the earlier instructions for the standard GCC build but rather than plain `make gcc`, please use:

    ```
    $ make intel
    ```

### 3.1.6 Installing without root access

Compiling and installing NumBAT itself does not rely on having root access to your machine. However, installing the supporting libraries such as SuiteSparse and Arpack is certainly simpler if you have root or the assistance of your system admin.

If this is not possible, you can certainly proceed by building and installing all the required libraries into your own tree within your home directory. It may be helpful to create a tree like the following so that the relevant paths mirror those of a system install

```
$HOME/
 |---my_sys/
      |---usr/
```

(continues on next page)

(continued from previous page)

```
            |---include/
            |---lib/
            |---bin/
```

### 3.1.7 Troubleshooting Linux installs

Performing a full build of NumBAT and all its libraries from scratch is a non-trivial task and it's possible you will hit a few stumbles. Here are a few traps to watch out for:

1. Please ensure to use relatively recent libraries for all the Python components. This includes using
   - Python: 3.11 or later
   - `matplotlib`: 3.9.0 or later
   - `scipy`: 1.13.0 or later
   - `numpy`: 2.0 or later
2. Be sure to follow the instructions above about setting up the virtual environment for NumBAT excusively. This will help prevent incompatible Python modules being added over time.
3. If you encounter an error about "missing symbols" in the NumBAT fortran module, there are usually two possibilities:
   - A shared library (a file ending in `.so`) is not being loaded correctly because it can't be found in the standard search path. To detect this, run `ldd nb_fortran.so` in the `backend/fortran` directory and look for any lines containing `not found`. (On MacOS, use `otools -L nb_fortran.so` instead of `ldd`.)

     You may need to add the directory containing the relevant libraries to your `LD_LIBRARY_PATH` in your shell setup files (eg. `~/.bashrc` or equivalent).
   - You may have actually encountered a bug in the NumBAT build process. Contact us for assistance as described in the introduction.
4. If NumBAT crashes during execution with a `Segmentation fault`, you have quite possibly found a bug that should be reported to us. Useful information about a crash can be obtained from the GNU debugger `gdb` as follows:
   1. Make sure that core dumps are enabled on your system. This article provides an excellent guide on how to do so.
   2. Ensure that `gdb` is installed.
   3. Rerun the script that causes the crash. You should now have a core dump in the directory determined in step 1.
   4. Execute `gdb` as follows:

      ```
      $ gdb <path_to_numbat_python_env>  <path_to_core file>
      ```

   5. In gdb, enter `bt` for *backtrace* and try to identify the point in the code at which the crash has occurred.

## 3.2 Installing on MacOS

NumBAT can also be installed on MacOS, though this is less tested than other versions. We are keen to increase our support of the MacOS build. Any comments on difficulties and solutions will be appreciated, so please don't hesitate to get in touch using the directions at *Seeking help with building NumBAT*.

The following steps have worked for us:

1. Open a terminal window on your desktop.

2. Ensure you have the Xcode Command Line Tools installed. This is the basic package for command line development on MacOS. If you do not or are not sure, enter the following command and then follow the prompts:

```
$ xcode-select --install
```

**Note** that there is a different version of the Xcode tools for each major release of MacOS. If you have upgraded your OS, say from Ventura to Sonoma, you must install the corresponding version of Xcode.

If the installer says Xcode is installed but an upgrade exists, you almost certainly want to apply that upgrade.

3. Make a folder for your NumBAT work in a suitable location in your folder tree. Then clone the github repository:

```
$ mkdir numbat
$ cd numbat
$ git clone https://github.com/michaeljsteel/NumBAT.git
$ cd NumBAT
```

This new `NumBAT` folder location is referred to as <NumBAT> in the following.

1. If it is not already on your system, install the MacPorts package manager using the appropriate installer for your version of MacOS at that page.
2. Install the Gmsh mesh generation tool. Just the main Gmsh installer is fine. The SDK and other features are not required.

   **Note:** After the installer has run, you must move the Gmsh application into your Applications folder by dragging the Gmsh icon into Applications.
3. Install a current gcc (we used gcc13):

```
$ sudo /opt/local/bin/port install gcc-devel
```

4. Install the Lapack and Blas linear algebra libraries:

```
$ sudo /opt/local/bin/port install lapack
```

5. Install the Arpack eigensolver:

```
$ sudo /opt/local/bin/port install arpack
```

6. Install the SuiteSparse matrix algebra suite:

```
$ sudo /opt/local/bin/port install suitesparse
```

7. Install a current python (we used python 3.12):

   Use the standard installer at https://www.python.org/downloads/macos/.

   (Note that this will install everything in */Library/Frameworks* and **not** override the System python in */System/Library/Frameworks.*)
8. Install python *virtualenv* package

```
$ cd /Library/Frameworks/Python.framework/Versions/3.12/bin/
$ ./python3.12 -m pip install --upgrade pip
$ ./pip3 install virtualenv
```

9. Create a NumBAT-specific python virtual environment in *~/nbpy3*

```
$ cd /Library/Frameworks/Python.framework/Versions/3.12/bin/
$ ./python3 -m virtualenv ~/nbpy3
```

10. Activate the new python virtual environment (note the leading fullstop)

```
$ . ~/nbpy3/bin/activate
```

11. Install necessary python libraries

```
$ pip3 install numpy matplotlib scipy psutil meson ninja
```

12. Check that the python installs work.

```
$ python3.12
>>> import matplotlib
>>> import numpy
>>> import scipy
>>> import psutil
```

13. Move to the NumBAT fortran directory:

```
$ cd backend/fortran
```

14. Move to the <NumBAT>/backend/fortran/ directory and open the file `meson.options` in a text editor. Check the values of the options in the `MacOS` section and change any of the paths in the `value` fields as required.

15. To start the build, enter:

```
$ make mac
```

16. If all is well, this will run to completion. If you encounter errors, please check that all the instructions above have been followed accurately. If you are still stuck, see sec-troubleshooting-linuxmacos-label for further ideas.

17. If you hit a compile error you can't resolve, please see the instructions at *Seeking help with building NumBAT* on how to seek help.

18. Once the build has apparently succeeded, it is time to test the installation with a short script that tests whether required applications and libraries can be found and loaded. Perform the following commands:

```
$ cd <NumBAT>/backend
$ python ./nb_install_tester.py
```

19. If this program runs without error, congratulations! You are now ready to proceed to the next chapter to begin using NumBAT.

20. Once again, if you run into trouble, please don't hesitate to get in touch for help using the instructions at *Seeking help with building NumBAT*. Please do send all the requested information, as it usually allows us to solve your problem faster.

### 3.2.1 Troubleshooting MacOS installs

Performing a full build of NumBAT and all its libraries from scratch is a non-trivial task and it's possible you will hit a few stumbles. Here are a few traps to watch out for:

1. Please ensure to use relatively recent libraries for all the Python components. This includes using
   - Python: 3.11 or later
   - `matplotlib`: 3.9.0 or later
   - `scipy`: 1.13.0 or later
   - `numpy`: 2.0 or later
2. Be sure to follow the instructions above about setting up the virtual environment for NumBAT excusively. This will help prevent incompatible Python modules being added over time.

3. In general, the GCC build is more tested and forgiving than the build with the Intel compilers and we recommend the GCC option. However, we do recommend using the Intel OneAPI math libraries as described above. This is the easiest way to get very high performance LAPACK and BLAS libraries with a well-designed directory tree.

4. If you encounter an error about "missing symbols" in the NumBAT fortran module, there are usually two possibilities:

   - A shared library (a file ending in `.so`) is not being loaded correctly because it can't be found in the standard search path. To detect this, run `ldd nb_fortran.so` in the `backend/fortran` directory and look for any lines containing `not found`. (On MacOS, use `otools -L nb_fortran.so` instead of `ldd`.)

     You may need to add the directory containing the relevant libraries to your `LD_LIBRARY_PATH` in your shell setup files (eg. `~/.bashrc` or equivalent).

   - You may have actually encountered a bug in the NumBAT build process. Contact us for assistance as described in the introduction.

## 3.3 Installing on Windows

There are three methods for installing NumBAT on Windows:

1. Use the pre-built binary installer.

   This is usually the easiest method to get going quickly. However, the pre-built installer is updated less frequently than the core github source code tree.

2. Build the entire system from scratch including all the mathematical support libraries.

   This method provides the most flexibility but does involve a significant number of steps. This is recommended for users with some experience of building code.

3. Build the core NumBAT code but obtain the supporting mathematical libraries with a pre-built installer.

   This is a good compromise for most users, allowing immediate access to new features in the github tree.

For all three methods, there are some common steps, including setting up the python environment and installing the open-source GMsh tool. These common steps are described in the next section and should be performed first. Then proceed to the section corresponding to your preferred installation mode.

### 3.3.1 Common steps - Setting up the NumBAT Python environment

NumBAT is entirely driven from Python, either as scripts or Jupyter notebooks, so a working Python installation is a basic requirement.

You can use an existing Python installation if you have a sufficiently recent version (>3.11.0) installed, or download a new Python installer. Note that NumBAT is a self-contained tree and will not add any files to your Python installation.

**Note: if you are using the pre-built NumBAT installer, for binary compatibility you *must* use Python version 3.13 and numpy version 2.2.**

1. If you do not have a suitable current Python, download and run the installer for a recent version from the Python website.

   By default, this will install Python in the directory `%HOMEPATH%\AppData\Local\Programs\Python\<PythonVer>`, say `%HOMEPATH%\AppData\Local\Programs\Python\Python312`.

2. Create a python virtual environment for working with NumBAT.

   You can use any name and location for your environment.

   To specify a virtual environment tree called *nbpy3*, open a command prompt (or your favourite Windows terminal app) from the Start Menu and enter

```
$ %HOMEPATH%\AppData\Local\Programs\Python\Python312\python.exe -m venv nbpy3
```

3. Activate the new python virtual environment

```
$ %HOMEPATH%\npby3\Scripts\activate
```

4. Install the necessary python tools and libraries

```
$ pip3 install numpy  matplotlib scipy psutil ninja meson==1.4.1
```

   Note that at last check, the most recent meson (1.5.0) is broken and we specify the earlier 1.4.1 version.

5. Finally, we will also need the Gnu `make` tool. This can be installed by typing

```
$ winget install ezwinports.make
```

   and then starting a new terminal (so that the PATH variable is updated to find `make.exe`.)

Now you can proceed to install NumBAT using one of the following methods:

1. Method 1: Using the complete pre-built installer in *Installing NumBAT Method 1: pre-built Windows installer*
2. Method 2: Building all components from source in *Method 2: Building all components from source*
3. Method 3: Building core NumBAT with pre-built mathematical libraries in *Method 3: Building core NumBAT with pre-built mathematical libraries*

### 3.3.2 Installing NumBAT Method 1: pre-built Windows installer

*Note*: The installer will allow you to run NumBAT problems and make changes to the code on the Python side only. You will not be able to make changes to the Fortran finite element code. (For most people, this is completely fine.) The installer is not updated as frequently as the main source tree.

**Note: the pre-built installer will only work with Python version 3.13. If you do not have this version installed, please follow the instructions for installing Python earlier in this section.**

We will actually run two installers: one for the free GMsh mesh generation tool, and one for NumBAT itself.

#### Setting up NumBAT's folder structure and installing gmsh

1. Choose a location for the base folder for building NumBAT and supporting libraries, say `c:\Users\<myname>\numbat`, which we will refer to as `<NumBAT_BASE>`.

   Create this folder using either Windows Explorer or a command prompt.

2. Use the Start Menu to open a Windows terminal.

3. In the terminal window, change to the `<NumBAT_BASE>` directory, then execute the following commands

```
$ mkdir nb_releases
$ mkdir usr_local
$ mkdir usr_local\packages
```

   Your NumBAT trees will be stored inside the *nb_releases* directory.

4. Download the Windows build of Gmsh and unzip the tree into `<NumBAT_Base>\usr_local\packages\gmsh`. The Gmsh executable should now be at `<NumBAT_Base>\usr_local\packages\gmsh\gmsh.exe`.

#### Installing the NumBAT binary installer

1. Download the Windows installer from the NumBAT github page. The link to the installer can be found at the bottom of the *Readme* section and also under the *Releases* heading in the right-hand column of the page.

   Be sure to download the *numbat_complete_installer_win64.exe* file and *not* the mathlibs installer.

2. Run the installer specifying a *new* install directory inside`` the `<NumBAT_BASE>\nb_releases` folder.

   We recommend including the NumBAT release number for the install directory name, for example `<NumBAT_BASE>\nb_releases\NumBAT-2.1.3`.

#### Testing the installation

1. In your Windows terminal, change directory to the newly installed NumBAT folder.
2. Activate your python environment and then move to the NumBAT examples/tutorials directory

```
$ %HOMEPATH%\npby3\Scripts\activate
$ cd examples\tutorials
```

   Remember, you must activate a python environment that uses Python 3.13.

3. You should now be able to run a NumBAT calculation

```
$ make tut01
```

4. If this program runs without error, congratulations! You are now ready to proceed to the next chapter to begin using NumBAT.

   If not, please check the instructions above again, and if still stuck, please read the *troubleshooting* section to attempt to diagnose the problem. Then follow the instructions at *Seeking help with building NumBAT* to seek assistance.

### 3.3.3 Method 2: Building all components from source

The Windows version of NumBAT is built using the native Windows toolchain including Visual Studio and the Intel Fortran compiler. Since a standard Windows installation is not set up as a development environment, there are a number of steps and tools required. If you are new to building software, you might like to begin with Method 3.

#### Windows build tools

We need to install a number of compilers and math libraries. The following tools are all free but will use several GB of disk space.

- ***Visual Studio*:**
  This is the primary Microsoft development environment.

  To install the free Community 2022 edition, download the main installer and follow the instructions.

- ***Intel OneAPI Base Toolkit*:**
  This is the main Intel developer environment including C/C++ compiler and many high performance math libraries.

  Download and run the installer accepting all defaults.

- ***Intel OneAPI HPC Toolkit***
  This adds the Intel Fortran compiler amongst other HPC tools.

  Download and run the installer accepting all defaults.

- ***Git***
  This is a widely-used source control tool that we use to download NumBAT and some other tools.

  Download and run the latest Git for Windows release, accepting all defaults.

  Some users may prefer to use a graphical interface such as GitHub Desktop. This is fine too.

1. ***GNU make***
   We will need the Gnu `make` tool. This can be installed by typing

```
$ winget install ezwinports.make
```

and then starting a new terminal (so that the PATH variable is updated to find `make.exe`.)

### Building the supporting math libraries

We can now build the supporting libraries, before proceeding to build NumBAT itself.

1. To build Arpack and SuiteSparse libraries, we need the *CMake* tool.

   This is a cross-platform build tool we will need for building some of the libraries.

   Download and run the latest release accepting all defaults.

2. Choose a location for the base directory for building NumBAT and supporting libraries, say `c:\Users\<myname>\numbat`, which we will refer to as `<NumBAT_BASE>`.

3. Use the Start Menu to open the *Intel OneAPI Command Prompt for Intel 64 for Visual Studio 2022*. This is simply a Windows terminal with some Intel compiler environment variables pre-defined.

4. In the terminal window, change to the `<NumBAT_BASE>` directory, then execute the following commands:

```
$ mkdir nb_releases
$ mkdir usr_local
$ mkdir usr_local\include
$ mkdir usr_local\lib
$ mkdir usr_local\packages
$ cd usr_local\packages
$ git clone https://github.com/opencollab/arpack-ng.git arpack-ng
$ git clone https://github.com/jlblancoc/suitesparse-metis-for-windows.git␣
↪suitesparse-metis
$ cd ..\..\nb_releases
$ git clone https://github.com/michaeljsteel/NumBAT.git nb_latest
```

5. Download the Windows build of Gmsh and unzip the tree into `usr_local\packages\gmsh`. The Gmsh executable should now be at `<NumBAT>\usr_local\packages\gmsh\gmsh.exe`.

6. Your `<NumBAT_BASE>` tree should now look like this:

```
%HOME%
    |---numbat
        |---nb_releases
        |---usr_local
            |---include
            |---lib
            |---packages
```

### Building SuiteSparse

This library performs sparse matrix algebra, used in the eigensolving routines of NumBAT.

1. In the Intel command terminal, cd to `<NumBAT_BASE>\usr_local\packages\suitesparse-metis`.

2. Enter the following command. It may take a minute or two to complete:

```
$ cmake -D WITH_MKL=ON -B build .
```

   Make sure to include the fullstop after `build`.

3. If that completes correctly, use Windows Explorer to open `<NumBAT_BASE>\usr_local\packages\suitesparse-metis\build\SuiteSparseProject.sln` with Visual Studio 2022.

4. In the pull-down menu in the ribbon, select the *Release* build. Then from the *Build* menu, select the *Build Solution* item to commence the build. This will take a couple of minutes.

5. Return to the command terminal and cd to <NumBAT_BASE>\usr_local. Then execute the following commands:

```
$ copy packages\suitesparse-metis\build\lib\Release\*.dll lib
$ copy packages\suitesparse-metis\build\lib\Release\*.lib lib
$ copy packages\suitesparse-metis\SuiteSparse\AMD\Include\*.h include
$ copy packages\suitesparse-metis\SuiteSparse\UMFPACK\Include\*.h include
$ copy packages\suitesparse-metis\SuiteSparse\SuiteSparse_config\*.h include
```

### Building Arpack-ng

This library performs an iterative algorithm for finding matrix eigensolutions.

1. In the Intel command terminal, cd to <NumBAT_BASE>\usr_local\packages\arpack-ng.
2. Enter the following command. It may take a minute or two to complete:

```
$ cmake -B build -T "fortran=ifx" -D CMAKE_BUILD_TYPE=Release -D BUILD_SHARED_
↪LIBS=OFF .
```

   Note the final fullstop!

3. If that completes correctly, use Windows Explorer to open <NumBAT_BASE>\usr_local\packages\arpack-ng\build\arpack.sln with Visual Studio 2022.
4. In the pull-down menu in the ribbon, select the *Release* build. Then from the *Build* menu select the *Build solution* option. This will take a few minutes.
5. Return to the command terminal and cd to <NumBAT_BASE>\usr_local. Then execute the following commands:

```
$ copy packages\arpack-ng\build\Release\* lib
$ copy packages\arpack-ng\ICB\*.h include
```

At long last, we are ready to build NumBAT itself.

### Building NumBAT

We are now ready to build and test NumBAT itself.

1. Your command prompt needs to have your NumBAT python environment activated.

   If you have opened a new prompt since setting up the python environment, enter

```
$ %HOMEPATH%\npby3\Scripts\activate
```

2. We need two additional python modules to run the build process

```
$ pip3 install ninja meson==1.4.1
```

   Note that at last check, the most recent meson (1.5.0) is broken and we specify the earlier 1.4.1 version.

3. In your command prmopt, move to your root <NumBAT_BASE> directory and then to the NumBAT folder itself:

```
$ cd <NumBAT_BASE>
$ cd nb_releases\nb_latest
```

   From this point, we refer to the current directory as <NumBAT>.

   In other words, <NumBAT> = <NumBAT_BASE>\nb_releases\nb_latest.

4. Setup the environment variables for the Intel compiler:

```
$  "c:\Program Files (x86)\Intel\oneAPI\setvars.bat"
```

5. Move to the <NumBAT>\backend\fortran directory and open the file meson.options in a text editor. Check the values of the options in the Windows section, particularly the value for windows_dir_nb_usrlocal and change any of the paths in the value fields as required.

6. To initiate the build, enter

```
$ make win
```

   This should take 2 to 3 minutes.

7. If all is well, this will run to completion. If you encounter errors, please check that all the instructions above have been followed accurately. If you are still stuck, see *Troubleshooting a Windows installation* for further ideas.

8. If you hit a compile error you can't easily resolve, please see the instructions at *Seeking help with building NumBAT* on how to seek help.

9. Copy the .dlls generated earlier to this directory:

```
$ copy ..\..\..\..\usr_local\lib\*.dll .
```

10. Also, copy the following Intel oneAPI .dlls to this directory:

```
$ copy "c:\Program Files (x86)\Intel\oneAPI\mkl\latest\bin\mkl_rt.2.dll" .
$ copy "c:\Program Files (x86)\Intel\oneAPI\mkl\latest\bin\mkl_intel_thread.2.dll
↪" .
$ copy "c:\Program Files (x86)\Intel\oneAPI\compiler\latest\bin\svml_dispmd.dll"␣
↪.
$ copy "c:\Program Files (x86)\Intel\oneAPI\compiler\latest\bin\libmmd.dll" .
$ copy "c:\Program Files (x86)\Intel\oneAPI\compiler\latest\bin\libifcoremd.dll"␣
↪.
$ copy "c:\Program Files (x86)\Intel\oneAPI\compiler\latest\bin\libifportMD.dll"␣
↪.
```

11. At this point, we are ready to test the installation with a short script that checks whether required applications and libraries can be found and loaded. Perform the following commands:

```
$ cd <NumBAT>/backend
$ python .\nb_install_tester.py
```

12. If this program runs without error, congratulations! You are now ready to proceed to the next chapter to begin using NumBAT. If not, please see the suggestions at *Troubleshooting a Windows installation*. But before moving on, please read the section *Creating a self-contained command terminal* on creating a specalised NumBAT command terminal.

13. Once again, if you run into trouble, please don't hesitate to get in touch for help using the instructions at *Seeking help with building NumBAT*. Please do send all the requested information, as it usually allows us to solve your problem faster.

### 3.3.4 Method 3: Building core NumBAT with pre-built mathematical libraries

1. Set up the Windows build environment.

   Follow the instructions in section *Windows build tools* to install *Visual Studio*, the *Intel OneAPI Base and HPC Toolkits*, *Git* and *GNU Make*.

2. Download the Windows maths support libraries for NumBAT installer from the NumBAT github page. The link to the installer can be found at the bottom of the *Readme* section and also under the *Releases* heading in the right-hand column of the page.

   Be sure to download the *numbat_mathlibs_installer_win64.exe* file and *not* the complete installer.

Run the installer choosing a base install directory of your choice, say `c:\Users\<myname>\numbat`, which we will refer to as the `<NumBAT_BASE>` folder.

3. Download the Windows build of Gmsh and unzip the tree into `<NumBAT_Base>\usr_local\packages\gmsh`. The Gmsh executable should now be at `<NumBAT_Base>\usr_local\packages\gmsh\gmsh.exe`.
4. Carry out the instructions in the section *Building NumBAT* to build and test the core NumBAT code.
5. You are now ready to start working with NumBAT.

   Before moving on, to make updating NumBAT simpler, please read the section *Creating a self-contained command terminal* on creating a specalised NumBAT command terminal.

### 3.3.5 Creating a self-contained command terminal

If you plan to build the fortran code frequently to keep up to date with changes in the source tree, both the python and Intel oneAPI paths need to be set up in your terminal. Doing this manually requires typing:

```
$ %HOMEPATH%\npby3\Scripts\activate
$ c:\Program Files (x86)\Intel\oneAPI\setvars.bat
```

This quickly becomes tedious. To automatically activate your python environment and ensure all other necessary paths are correctly setup, it is helpful to create a dedicated launcher for the desktop that executes the required commands on first opening the terminal.

Here is a procedure for doing this

1. Copy the launcher file `numbat_cmd.bat` to your NumBAT root directory:

   ```
   $ copy <NumBAT>\share\numbat_cmd.bat <NumBAT_BASE>
   ```

2. Create a desktop shortcut to the Windows command terminal by using File Explorer to open the folder `c:\Windows\System32`, typing `cmd.exe` in the search box at top right, and then right-clicking *Send to Desktop*.
3. Right click on the new shortcut and open its *Properties* dialog.
4. Select the *General* tab and change the name field at the top to *NumBAT Terminal*.
5. Select the *Shortcut* tab and change the *Target* field to `%windir%\System32\cmd.exe "/K" %HOMEPATH%\numbat\numbat_cmd.bat`
6. Click the *Change Icon* button and select the NumBAT icon at `<NumBAT>\docs\source\numbat.ico`.

### 3.3.6 Troubleshooting a Windows installation

1. My build of NumBAT completes but the `nb_install_tester.py` program complains the NumBAT fortran `nb_fortran.pyd` dynamically linked library (DLL) can't be loaded.

   This is usually due to another DLL required by NumBAT not being found, either because it is in an unexpected location or missing altogether. This can be a little painful to diagnose. The following procedure is relatively straightforward.

   1. Download the *Dependencies* tool available as a zip file install from github. This tool displays all the DLL dependencies of a given file and whether or not they have been located in the file system. Extract the zip file to a folder named `dependencies` in `<NumBAT_BASE>\usr_local\packages`.
   2. Now we can apply the tool to the NumBAT python dll.

   Start the `DependenciesGui.exe` tool:

   ```
   $ <NUMBAT_BASE>\usr_local\packages\dependencies\DependenciesGUI.exe
   ```

   Browse to your NumBAT folder and open `backend\fortran\nb_fortran.pyd`.

   Examine the output and note any red highlighted entries. These indicate required DLLs that have not been found. If one or more such lines appear, read through the install instructions again and ensure that any commands to copy DLLs to particular locations have been executed.

1. Alternatively, you can try the command line version of this tool

```
$ <NUMBAT_BASE>\usr_local\packages\dependencies\Dependencies.exe -depth 5 -
↪modules nb_fortran.pyd
```

If you find a missing DLL by one of these methods, please *let us know*. It may suggest a problem with the pre-built installer or the documentation instructions.

### 3.3.7 Installing the Linux version via a Virtual Machine

Yet another way to run NumBAT on Windows or MacOS is by installing Ubuntu as a virtual machine using either Microsoft Hyper-V or Oracle Virtual Box, or a similar tool on MacOS.

Then NumBAT can be installed using exactly the same procedure as described above for standard Linux installations.

As the native installation methods now work well on all platforms, this is not recommended for normal use.

## 3.4 Seeking help with building NumBAT

If you are having trouble building NumBAT or are experiencing crashes, we will do our best to assist you.

Before writing for help (see contact details in *Introduction to NumBAT*), please do the following:

1. Download the latest version of NumBAT from github and ensure the problem remains.
2. If on Linux or MacOS, run the script `./backend/nb_runconfig_test.sh` from the main NumBAT directory.

   This will create the file `./nb_buildconfig.txt` in the main directory with useful details about your build configuration and environment.

   *If on Windows*, do the same using the file `.\backend\nb_runconfig_test.bat` instead.
3. In your email, indicate the date on which you last downloaded NumBAT from github.
4. Attach the `nb_buildconfig.txt` file.
5. If the problem is a crash while running a NumBAT script, please attach the python script file and a description of how to observe the problem.
6. If you are on Linux and encounter a segfault or "Segmentation violation", it is especially helpful if you able to follow the instructions under *Troubleshooting Linux installs* to generate a debugging trace with GDB and send a screen shot of the trace.

## 3.5 Building the documentation

If you should want to, you can rebuild the documentation you are currently reading. You will need a working LaTeX installation to build the pdf version.

### 3.5.1 Steps for building the documentation

- Ensure that all examples have been built.

  Most of the figures will only be available after you have run all the problems in the `tutorial`, `lit_ex` and `josab_tutorial` sub-directories of the `examples` directory have been run. This can be done by running `make` in each of those directories. Be aware that some of these problems are quite large and may require some time to complete depending on your computer's performance.

- Install documentation related python modules.

  Activate your normal NumBAT python environment and then type

```
$ pip3 install sphinx nbsphinx sphinx_subfigure sphinxcontrib-bibtex setuptools␣
↪pandoc ipython pygments
```

- Install Pandoc.

  You may also need to install the Pandoc package for your distribution using your package manager or from https://pandoc.org/ .

- Build the docs (choosing whichever version you require)

```
$ cd <numbat_base>/docs
$ make html
$ make latexpdf
```

- The output will be found in the *<numbat_base>docsbuild* directory.

CHAPTER

FOUR

# BASIC USAGE

## 4.1 A First Calculation

We're now ready to start using NumBAT.

Let's jump straight in and run a simple calculation. Later in the chapter, we go deeper into some of the details that we will encounter in this first example.

### 4.1.1 Tutorial 1 – Basic SBS Gain Calculation

Simulations with NumBAT are generally carried out using a python script file.

This example, contained in `<NumBAT>examples/tutorials/sim-tut_01-first_calc.py` calculates the backward SBS gain for a rectangular silicon waveguide surrounded by air.

Move into the examples/tutorials directory and then run the script by entering:

```
$ python3 sim-tut_01-first_calc.py
```

After a short while, you should see some values for the SBS gain printed to the screen. In many more examples/tutorials in the subsequent chapters, we will meet much more convenient forms of output, but for now let's focus on the steps involved in this basic calculation.

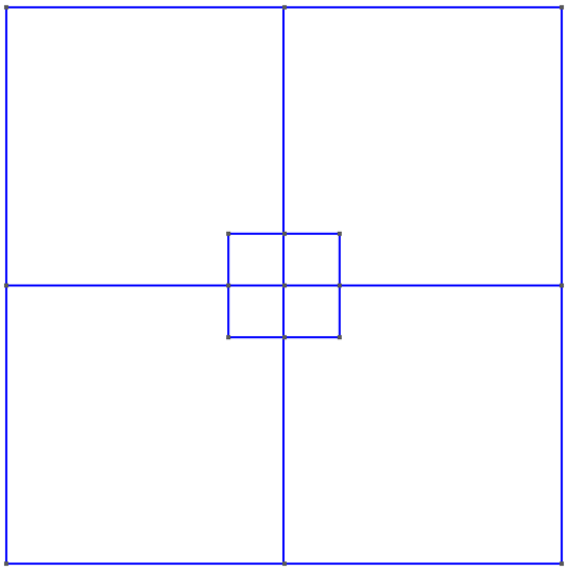
The sequence of operations (annotated in the source code below as Step 1, Step 2, etc) is:

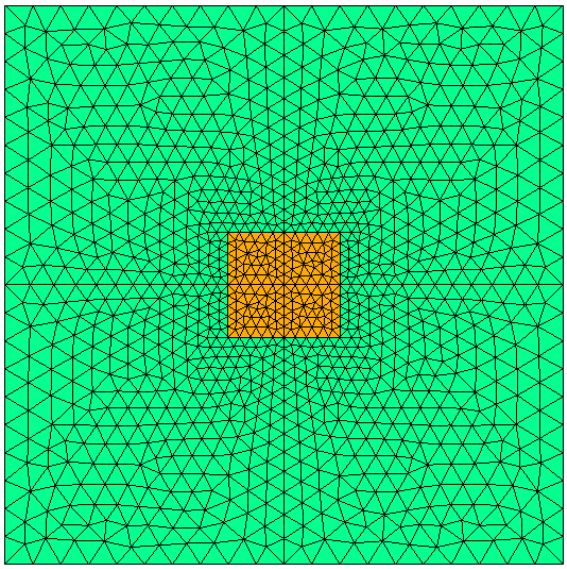
1. Add the NumBAT install directory to Python's module search path and then import the NumBAT python modules.
2. Set parameters to define the structure shape and dimensions.
3. Set parameters determining the range of electromagnetic and elastic modes to be solved.
4. Create the primary `NumBATApp` object to access most NumBAT features and set the filename prefix for all outputs.
5. Construct the waveguide with `nbapp.make_structure` out of a number of `materials.Material` structure.
6. Generate output files containing images of the finite element mesh and final refractive index. These are illustrated in figures below.
7. Solve the electromagnetic problem at a given *free space* wavelength $\lambda$. The function `modecalcs.calc_EM_modes()` returns an `EMSimResult` object containing electromagnetic mode profiles, propagation constants, and potentially other data which can be accessed through various methods we will meet in later examples/tutorials. The calculation is provided with a rough estimate of the effective index to guide the solver the find guided eigenmodes in the desired part of the spectrum. After the calculation, we can obtain the exact effective index of the fundamental mode using `modecalcs.neff()`.
8. Display the propagation constants in units of $\mathrm{m}^{-1}$ of the EM modes using `modecalcs.kz_EM_all()`
9. Calculate the electromagnetic fields for the Stokes mode. As the pump and Stokes frequencies are very similar, the Stokes modes can be found with high precision by a simple complex conjugate transformation of the pump fields.

10. Identify the desired elastic wavenumber from the difference of the pump and Stokes propagation constants and solve the elastic problem. `modecalcs.calc_AC_modes()` returns an `ACSimResult` object containing the elastic mode profiles, frequencies and potentially other data at the specified propagation constant `k_AC`.
11. Display the elastic frequencies in Hz using `modecalcs.nu_AC_all()`.
12. Use `integration.get_gains_and_qs()` to generate a `GainProps` object containing information on the total SBS gain, contributions from photoelasticity and moving boundary effects, and the elastic loss.
13. Extract desired values from the gain properties and print them to the screen.

You may have noticed from this description that the eigenproblems for the electromagnetic and acoustic problems are framed in opposite senses. The electromagnetic problem finds the wavenumbers $k_{z,n}(\omega)$ (or equivalently the effective indices) of the modes at a given free space wavelength (ie. at a specified frequency $\omega = 2\pi c/\lambda$). The elastic solver, however, works in the opposite direction, finding the elastic modal frequencies $\nu_n(q_0)$ at a given elastic propagation constant $q_0$. While this might seem odd at first, it is actually the natural way to frame SBS calculations.

We emphasise again, that for convenience, the physical dimensions of waveguides are specified in nanometres. All other quantities in NumBAT are expressed in the standard SI base units.





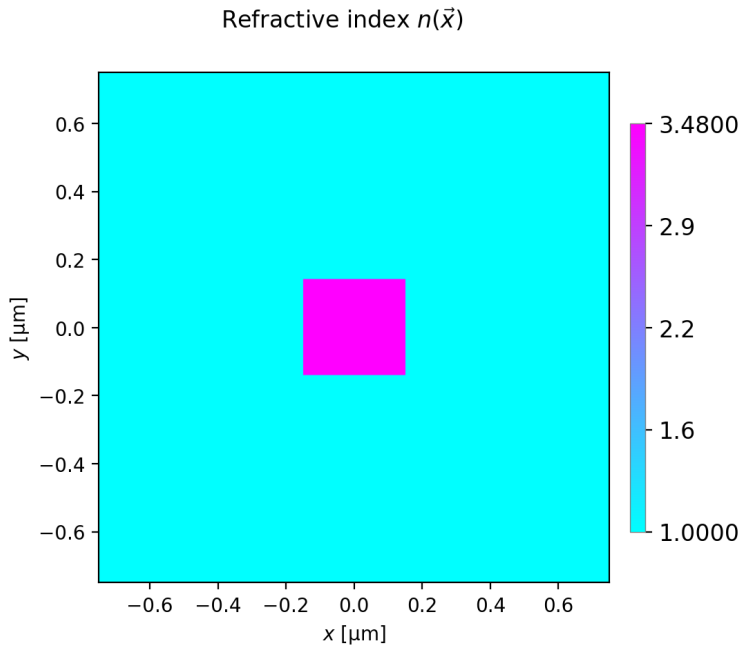



Fig. 1: Generated meshes and refractive index profile.

Here's the full source code for this tutorial:

```
print(nbapp.final_report())
""" Calculate the backward SBS gain for modes in a
    silicon waveguide surrounded in air.
"""

# Step 1
import sys
import numpy as np

from pathlib import Path
sys.path.append(str(Path('../../backend')))


import numbat
import integration

import materials


# Naming conventions
# AC: acoustic
# EM: electromagnetic
# q_AC: acoustic wavevector

print('\n\nCommencing NumBAT tutorial 1')

# Step 2
# Geometric Parameters - all in nm.

lambda_nm = 1550.0  # Wavelength of EM wave in vacuum.
```

(continues on next page)

(continued from previous page)

```
# Waveguide widths.
inc_a_x = 300.0
inc_a_y = 280.0

# Unit cell must be large to ensure fields are zero at boundary.
domain_x = 1500.0
domain_y = domain_x


# Shape of the waveguide.
inc_shape = 'rectangular'

# Step 3
# Number of electromagnetic modes to solve for.
num_modes_EM_pump = 20
num_modes_EM_Stokes = num_modes_EM_pump

# Number of acoustic modes to solve for
num_modes_AC = 20

# The EM pump mode(s) for which to calculate interaction with AC modes.
# Can specify a mode number (zero has lowest propagation constant) or 'All'.
EM_mode_index_pump = 0
# The EM Stokes mode(s) for which to calculate interaction with AC modes.
EM_mode_index_Stokes = 0
# The AC mode(s) for which to calculate interaction with EM modes.
AC_mode_index = 'All'

# Step 4
# Create the primary NumBAT application object and set the file output prefix
prefix = 'tut_01'
nbapp = numbat.NumBATApp(prefix)

# Step 5
# Use specified parameters to create a waveguide object.
# to save the geometry and mesh as png files in backend/fortran/msh/

wguide = nbapp.make_structure(inc_shape, domain_x, domain_y, inc_a_x, inc_a_y,
                              material_bkg=materials.make_material("Vacuum"),
                              material_a=materials.make_material("Si_2016_Smith"),
                              lc_bkg=.05,  # in vacuum background
                              lc_refine_1=2.5,  # on cylinder surfaces
                              lc_refine_2=2.5)  # on cylinder center

# Step 6
# Optionally output plots of the mesh and refractive index distribution
wguide.plot_mesh(prefix)
wguide.plot_refractive_index_profile(prefix)

# Step 7
# Calculate the Electromagnetic modes of the pump field.

# We provide an estimated effective index of the fundamental guided mode to steer the␣
↪solver.
n_eff = wguide.get_material('a').refindex_n-0.1
```

(continues on next page)

(continued from previous page)

```
sim_EM_pump = wguide.calc_EM_modes(num_modes_EM_pump, lambda_nm, n_eff)

# Report the exact effective index of the fundamental mode
n_eff_sim = np.real(sim_EM_pump.neff(0))
print("\n Fundamental optical mode ")
print(" n_eff = ", np.round(n_eff_sim, 4))


# Step 8
# Display the wavevectors of EM modes.
v_kz = sim_EM_pump.kz_EM_all()
print('\n k_z of electromagnetic modes [1/m]:')
for (i, kz) in enumerate(v_kz):
    print(f'{i:3d}  {np.real(kz):.4e}')

# Step 9
# Calculate the Electromagnetic modes of the Stokes field.
# For an idealised backward SBS simulation the Stokes modes are identical
# to the pump modes but travel in the opposite direction.
sim_EM_Stokes = sim_EM_pump.clone_as_backward_modes()

# Alternatively, solve again directly
# sim_EM_Stokes = wguide.calc_EM_modes(lambda_nm, num_modes_EM_Stokes, n_eff,␣
↪Stokes=True)


# Step 10
# Calculate Acoustic modes, using the mesh from the EM calculation.

# Find the required acoustic wavevector for backward SBS phase-matching
q_AC = np.real(sim_EM_pump.kz_EM(0) - sim_EM_Stokes.kz_EM(0))

print('\n Acoustic wavenumber (1/m) = ', np.round(q_AC, 4))

sim_AC = wguide.calc_AC_modes(num_modes_AC, q_AC, EM_sim=sim_EM_pump)

# Step 11
# Print the frequencies of AC modes.
v_nu = sim_AC.nu_AC_all()
print('\n Freq of AC modes (GHz):')
for (i, nu) in enumerate(v_nu):
    print(f'{i:3d}  {np.real(nu)*1e-9:.5f}')


# Step 12

# Do not calculate the acoustic loss from our fields, instead set a Q factor.
set_q_factor = 1000.

# Calculate interaction integrals and SBS gain for PE and MB effects combined,
# as well as just for PE, and just for MB. Also calculate acoustic loss alpha.


gain = integration.get_gains_and_qs(
    sim_EM_pump, sim_EM_Stokes, sim_AC, q_AC, EM_mode_index_pump=EM_mode_index_pump,
    EM_mode_index_Stokes=EM_mode_index_Stokes, AC_mode_index=AC_mode_index, fixed_
```

(continues on next page)

(continued from previous page)

```
↪Q=set_q_factor)

# Step 13
# SBS_gain_tot, SBS_gain_PE, SBS_gain_MB are 3D arrays indexed by pump, Stokes and␣
↪acoustic mode
# Extract those of interest as a 1D array:

SBS_gain_PE_ij = gain.gain_PE_all_by_em_modes(EM_mode_index_pump, EM_mode_index_
↪Stokes)
SBS_gain_MB_ij = gain.gain_MB_all_by_em_modes(EM_mode_index_pump, EM_mode_index_
↪Stokes)
SBS_gain_tot_ij = gain.gain_total_all_by_em_modes(EM_mode_index_pump, EM_mode_index_
↪Stokes)

# Print the Backward SBS gain of the AC modes.
print("\nContributions to SBS gain [1/(WM)]")
print("Acoustic Mode number | Photoelastic (PE) | Moving boundary(MB) | Total")

for (m, gpe, gmb, gt) in zip(range(num_modes_AC), SBS_gain_PE_ij, SBS_gain_MB_ij, SBS_
↪gain_tot_ij):
    print(f'{m:8d}  {gpe:18.6e}  {gmb:18.6e}  {gt:18.6e}')


# Mask negligible gain values to improve clarity of print out.
threshold = 1e-3
masked_PE = np.where(np.abs(SBS_gain_PE_ij) > threshold, SBS_gain_PE_ij, 0)
masked_MB = np.where(np.abs(SBS_gain_MB_ij) > threshold, SBS_gain_MB_ij, 0)
masked_tot = np.where(np.abs(SBS_gain_tot_ij) > threshold, SBS_gain_tot_ij, 0)

print("\n Displaying gain results with negligible components masked out:")

print("AC mode | Photoelastic (PE) | Moving boundary(MB) | Total")
for (m, gpe, gmb, gt) in zip(range(num_modes_AC), masked_PE, masked_MB, masked_tot):
    print(f'{m:8d}  {gpe:12.4f}  {gmb:12.4f}  {gt:12.4f}')

print(nbapp.final_report())
```

In the next few chapters, we meet many more examples that show the different capabilities of NumBAT and provided comparisons against analytic and experimental results from the literature.

For the remainder of this chapter, we will explore some of the details involved in specifying a wide range of waveguide structures.

## 4.2 General Simulation Procedures

Simulations with NumBAT are generally carried out using a python script file. This file is kept in its own directory which may or may not be within your NumBAT tree. All results of the simulation are automatically created within this directory. This directory then serves as a complete record of the calculation. Often, we will also save the simulation structure within this directory for future inspection, manipulation, plotting, etc.

These files can be edited using your choice of text editor (for instance `nano` or `vim`) or an IDE (for instance MS Visual Code or `pycharm`) which allow you to run and debug code within the IDE.

To save the results from a simulation that are displayed upon execution (the print statements in your script) use:

```
$ python3 ./sim-tut_01-first_calc.py | tee log-simo.log
```

To have direct access to the simulation structure upon the completion of a script use:

```
$ python3 -i ./sim-tut_01-first_calc.py
```

This will execute the python script and then return you into an interactive python session within the terminal. This terminal session provides the user experience of an ipython type shell where the python environment and all the simulation structure are in the same state as in the script when it has finished executing. In this session you can access the docstrings of structure, classes and methods. For example:

```
>>> from pydoc import help
>>> help(structure.Structure)
```

where we have accessed the docstring of the Struct class from `structure.py`.

## 4.3 Script Structure

As with our first example above, most NumBAT scripts proceed with a standard structure:

- importing NumBAT modules
- defining materials
- defining waveguide geometries and associating them with material properties
- solving electromagnetic and acoustic modes
- calculating gain and other derived quantities

CHAPTER

FIVE

# TUTORIAL

This chapter provides a first set of graded examples/tutorials for learning NumBAT and exploring its applications. Before attempting your own calculations with NumBAT, we strongly advise working through the sequence of tutorial exercises which are largely based on literature results.

We will meet a significant number of NumBAT functions in these examples/tutorials, though certainly not all. The full Python interface is documented in the section *Python Interface API*.

You may then choose to explore relevant examples drawn from a recent tutorial paper by Dr Mike Smith and colleagues, and a range of other literature studies, which are provided in the subsequent chapters, *JOSA-B Tutorial Paper* and *Additional Literature Examples*.

Those chapters include examples of validating NumBAT against literature results and analytic solutions where possible.

## 5.1 Some Key Symbols

As far as practical we use consistent notation and symbols in the tutorial files. The following list introduces a few commonly encountered ones. Note that with the exception of the free-space wavelength $\lambda$ and the spatial dimensions of waveguide structures, which are both specified in nanometres (nm), all quantities in NumBAT should be expressed in the standard SI units. For example, elastic frequencies $\nu$ are expressed in Hz, not GHz.

**`lambda_nm`**
: This is the *free-space* optical wavelength $\lambda$ satisfying $\lambda = 2\pi c/\omega$, where $c$ is the speed of light and $\omega$ is the angular frequency. **For convenience, this parameter is specified in nm.**

  For most examples, we use the conventional value $\lambda$ =1550 nm.

**`omega, omega_EM, om_EM`**
: This is the electromagnetic *angular* frequency $\omega = 2\pi c/\lambda$ specified in $\mathrm{rad.s}^{-1}$.

**`k, beta, k_EM`**
: This is the electromagnetic *wavenumber* or *propagation constant* $k$ or $\beta$, specified in $\mathrm{m}^{-1}$.

**`neff, n_eff`**
: This is the electromagnetic modal *effective index* $\bar{n} = ck/\omega$, which is dimensionless.

**`nu, nu_AC`**
: This is the acoustic frequency $\nu$ specified in Hz.

**`Omega, Omega_AC, Om_AC`**
: This is the acoustic *angular* frequency $\Omega = 2\pi\nu$ specified in $\mathrm{rad.s}^{-1}$.

**`q, q_AC`**
: This is the acoustic *wavenumber* or *propagation constant* $q = v_{ac}\Omega$, where $v_{ac}$ is the phase speed of the wave. The acoustic wavenumber is specified in $\mathrm{m}^{-1}$.

**`m`**
: This is an integer corresponding to the mode number $m$ of an electromagnetic mode $\vec{E}_m(\vec{r})$ or an acoustic mode $\vec{u}_m(\vec{r})$.

For both electromagnetic and acoustic modes, counting of modes begins with `m=0` and are ordered by decreasing effective index and increasing frequency respectively.

For the electromagnetic problem in which frequency/free-space wavelength is the independent variable, the $m = 0$ mode has the *highest* effective index $\bar{n}$ and *highest* wavenumber $k$ of any mode for a given angular frequency $\omega$.

For the acoustic problem, the wavenumber $q$ is the independent variable and we solve for frequency $\nu = \Omega/(2\pi)$. The $m = 0$ mode has the *lowest* frequency $\nu$ of any mode for a given wavenumber $q$.

The integer $m$ therefore has no particular correspondence to the conventional two index mode indices for fibre or rectangular waveguides.

**`inc_a_x, inc_a_y, inc_b_x, inc_b_y, slab_a_x, slab_a_y, ... etc`**
These are dimensional parameters specifying the lengths of different aspects of a given structure: rib height, fibre radius etc. **For convenience, these parameters are specified in nm.**

## 5.2 Elementary Tutorials

We now walk through a number of simple simulations that demonstrate the basic use of NumBAT located in the <NumBAT>/examples/tutorials directory.

### 5.2.1 Tutorial 2 – SBS Gain Spectra

The first example we met in the previous chapter only printed numerical data to the screen with no graphical output. This example, contained in <NUMBAT>/examples/tutorials/sim-tut_02-gain_spectra-npsave.py considers the same silicon-in-air structure but adds plotting of fields, gain spectra and techniques for saving and reusing data from earlier calculations.

As before, move to the <NUMBAT>/examples/tutorials directory, and then run the calculation by entering:

```
$ python3 sim-tut_02-gain_spectra-npsave.py
```

Or you can take advantage of the Makefile provided in the directory and just type:

```
$ make tut02
```

Some of the tutorial problems can take a little while to run, especially if your computer is not especially fast. To save time, you can run most problems with a coarser mesh at the cost of somewhat reduced accuracy, by adding the flag fast=1 to the command line:

```
$ python3 sim-tut_02-gain_spectra-npsave.py fast=1
```

Or using the makefile technique, simply

```
$ make ftut02
```

The calculation should complete in a minute or so. You will find a number of new files in the current directory beginning with the prefix tut_02 (or ftut_02 if you ran in fast mode).

#### Gain Spectra

The Brillouin gain spectra and plotted using the functions integration.get_gains_and_qs() and GainProps.plot_spectra(). The results are contained in the file tut_02-gain_spectra.png which can be viewed in any image viewer. On Linux, for instance you can use

```
$ eog tut_02_gain_spectra.png
```

to see this image:

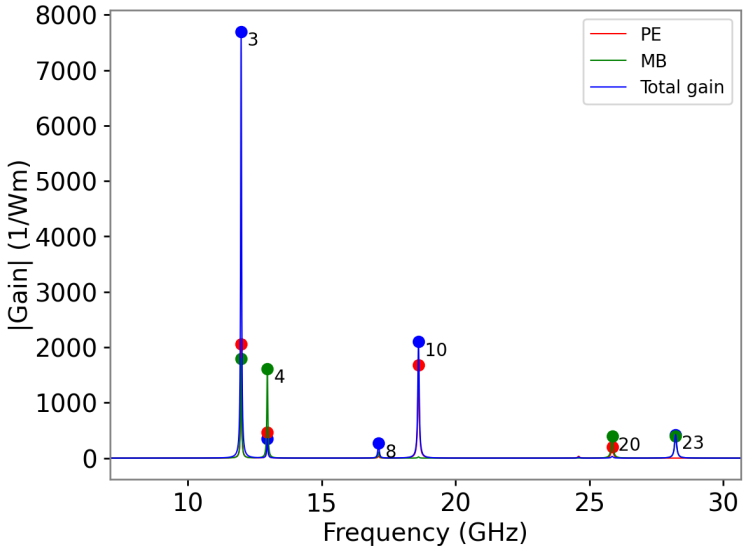



Fig. 1: Gain spectrum in `tut_02-gain_spectra.png` showing gain due to the photoelastic effect, gain due to moving boundary effect, and the total gain. The numbers near the main peaks identify the acoustic mode associated with the resonance.

Note how the different contributions from the photoelastic and moving-boundary effects are visible. In some cases, the total gain (blue) may be less than one or both of the separate effects if the two components act with opposite sign. (This is because the different contributions to the gain add as complex amplitudes). *JOSA-B Tutorial Paper* and *Additional Literature Examples*. (See Literature example 1 in the chapter *Additional Literature Examples* for an interesting cexample of this phenomenon.)

Note also that prominent resonance peaks in the gain spectrum are labelled with the mode number $m$ of the associated acoustic mode. This makes it easy to find the spatial profile of the most relevant modes (see below).

### Mode Profiles

The choice of parameters for `plot_gain_spectra()` has caused several other files to be generated showing a zoomed-in version near the main peak, and the whole spectrum on log and dB scales:

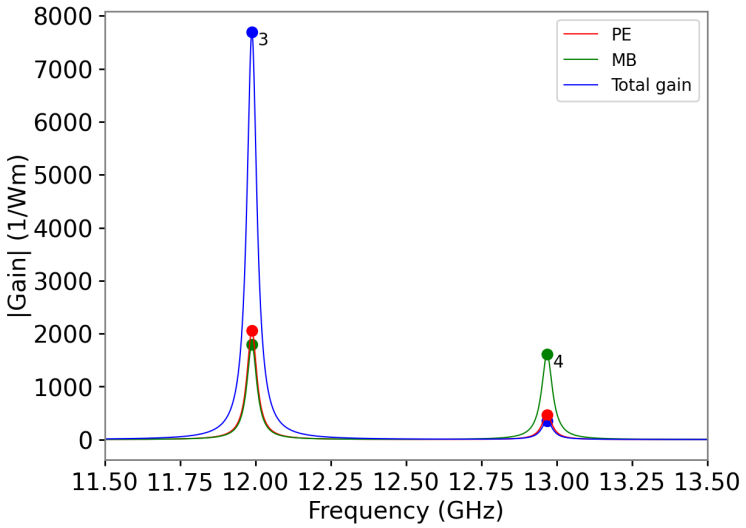



Fig. 2: Zoom-in of the gain spectrum in the previous figure in the file `tut_02-gain_spectra_zoom.png` .

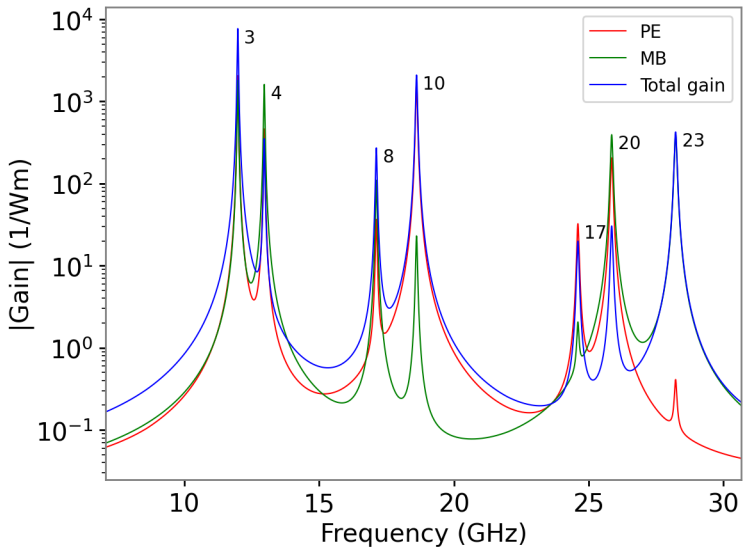



Fig. 3: Gain spectrum viewed on a log scale in the field `tut_02-gain_spectra-logy.png` .

This example has also generated plots of some of the electromagnetic and acoustic modes that were found in solving the eigenproblems. These are created using the calls to `plot_modes()` and stored in the sub-directory `tut_02-fields`.

Note that a number of useful parameters are also displayed at the top-left of each mode profile. These parameters can also be extracted using a range of function calls on a `Mode` object (see the API docs). Observe that NumBAT chooses the phase of the mode profile such that the transverse components are real. Note that the $E_z$ component is $\pi/2$ out of phase with the transverse components. (Since the structure is lossless, the imaginary parts of the transverse field, and the real part of $E_z$ are zero). The same is true for the magnetic field components and the elastic displacement fields.

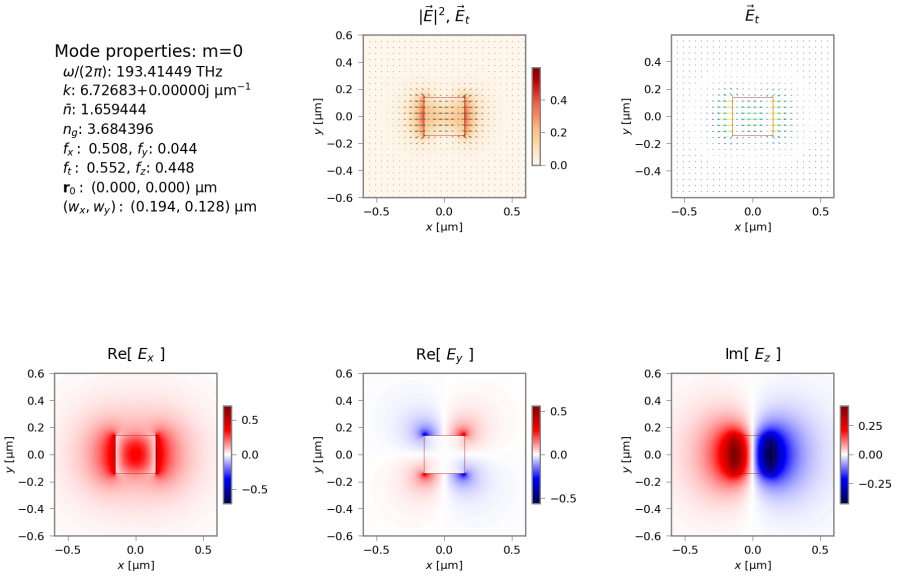



Fig. 4: Electric field profile of the fundamental ($m = 0$) optical mode profile stored in `tut_02-fields/EM_E_mode_00.png`. The plots show the modulus of the whole electric field $|\vec{E}|^2$, a vector plot of the transverse field $\vec{E}_t = (E_x, E_y)$, and the three components of the electric field.

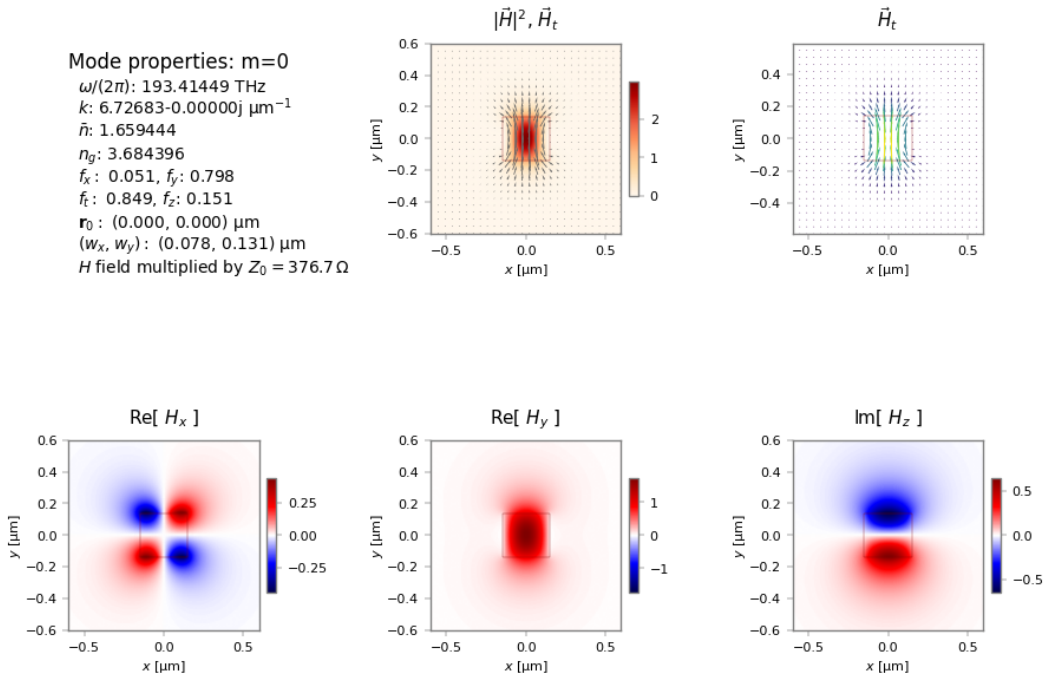



Fig. 5: Magnetic field profile of the fundamental ($m = 0$) optical mode profile showing modulus of the whole magnetic field $|\vec{H}|^2$, vector plot of the transverse field $\vec{H}_t = (H_x, H_y)$, and the three components of the magnetic field.

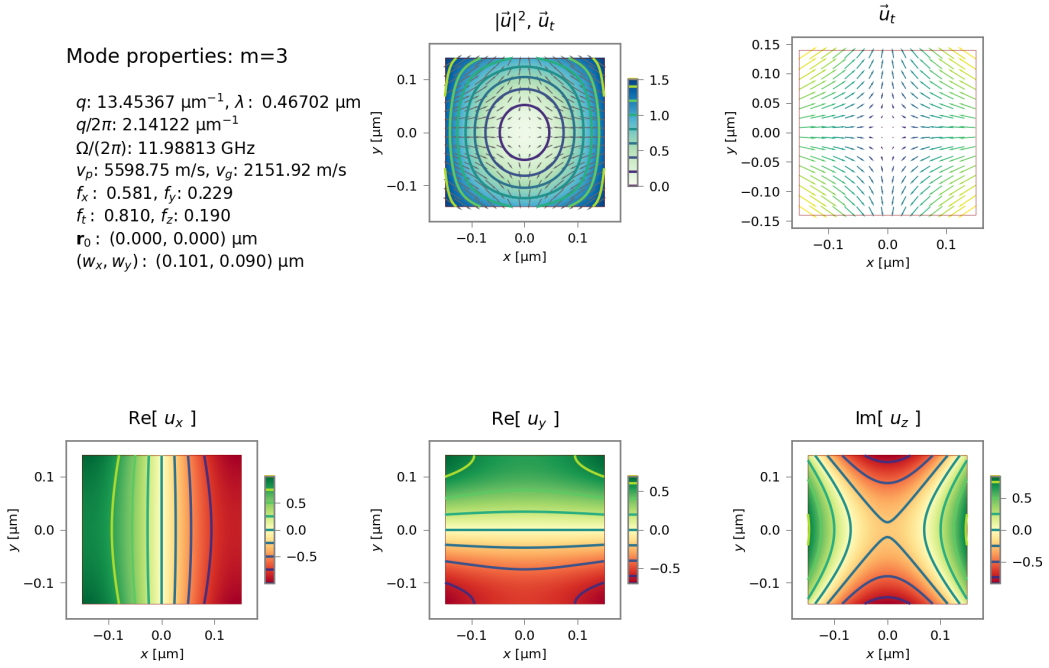



Fig. 6: Displacement field $\vec{u}(\vec{r})$ of the $m = 3$ acoustic mode with gain dominated by the moving boundary effect (green curve in gain spectra). As with the optical fields, the $u_z$ component is $\pi/2$ out of phase with the transverse components. Note that the frequency of $\Omega/(2\pi) = 11.99$ GHz (listed in the upper-left corner) corresponds to the first peak in the gain spectrum.

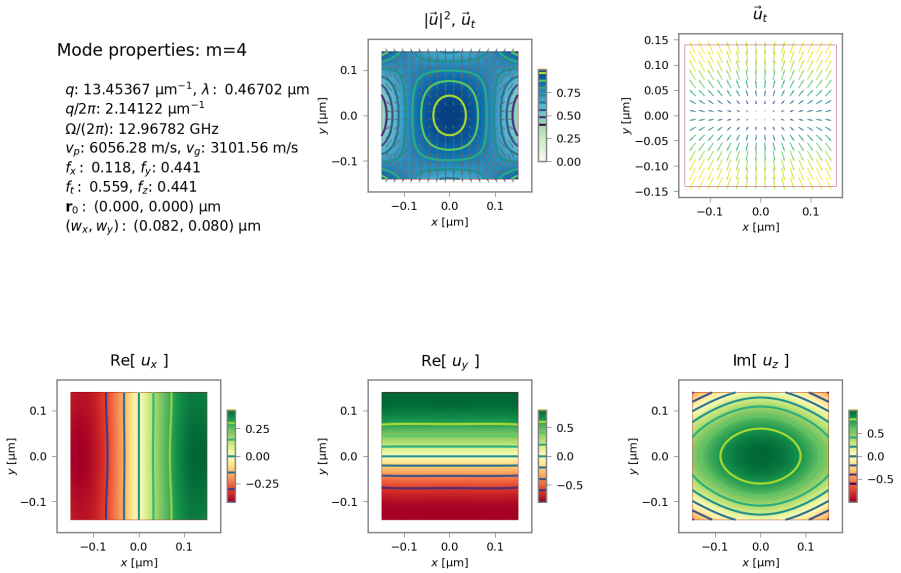



Fig. 7: Displacement field $\vec{u}(\vec{r})$ of the $m = 4$ acoustic mode with gain dominated by the photo-elastic effect (red curve in gain spectra). Note that the frequency of $\Omega/(2\pi) = 13.45$ GHz corresponds to the second peak in the gain spectrum.

A number of plot settings including colormaps and font sizes can be controlled using the `numbat.toml` file. This is discussed in *Additional details*.

### Miscellaneous comments

Here are some further elements to note about this example:

1. When using the `fast=` mode, the output data and fields directory begin with `ftut_02` rather than `tut_02`.
2. It is frequently useful to be able to save and load the results of simulations to adjust plots without having to repeat the entire calculation. Here the flag `reuse_old_fields` determines whether the calculation should be done afresh and use previously saved data. This is performed using the `save_simulation()` and `load_simulation()` calls.
3. Plots of the modal field profiles are obtained using the `plot_modes` methods of the EM and elastic sim result structure. Both electric and magnetic fields can be selected using `EM_E` or `EM_H` as the value of the `field_type` argument. The selection of mode numbers to be plotted is specified by `ivals`. These fields are stored in a folder `tut_02-fields/` within the tutorial folder. Later we will see how an alternative approach in which we extract a `Mode` object from a `Simulation` which represents a single mode that is able to plot itself. This can be more convenient.
4. The overall amplitude of the modal fields is arbitrary. In NumBAT, the maximum value of the electric field is normalised to be 1.0, and this may be interpreted as a quantity in units of V/m, $\sqrt{\mathrm{W}}$ or other units as desired. Importantly, when plotted, the *magnetic* field $\vec{H}(\vec{r})$ is multiplied by the impedance of free space $Z_0 = \sqrt{\mu_0/\epsilon_0}$ so that the plotted quantities $Z_0\vec{H}(\vec{r})$ and $\vec{E}(\vec{r})$ *have the same units*, and the relative amplitudes of the electric and magnetic field plots can be compared meaningfully.
5. The `suppress_imimre` option suppresses plotting of the $\mathrm{Im}[F_x]$, $\mathrm{Im}[F_y]$ and $\mathrm{Re}[F_z]$ components of the fields $\vec{F} \in [\vec{E}, \vec{H}, \vec{u}]$. In a lossless non-leaky problem, these fields should normally be zero at all points and therefore not useful to plot.
6. By default, plots are exported as `png` format. This can be adjusted in your `numbat.toml` plot settings file.
7. Plots of both spectra and modes are generated with a best attempt at font sizes, line widths etc, but the range of potential cases make it impossible to find a selection that works in all cases, and you can use the `numbat.toml` file to fine tune your plots. Further, some plot functions support the passing of a `plotting.Decorator` object that can vary the settings of some parameters and also pass additional commands to write on the plot axes. This should be regarded as a relatively advanced NumBAT feature.

8. Vector field plots often require tweaking to get an attractive set of vector arrows. The `quiver_points` option controls the number of arrows drawn along each direction. Other settings can be controlled in your `numbat.toml` plot settings file.
9. The plot functions and the `Decorator` class support many options. Consult the API chapter for details on how to fine tune your plots.

### 5.2.2 Tutorial 3a – Investigating Dispersion and np.save/np.load

This example, contained in `examples/tutorials/sim-tut_03_1-dispersion-npload.py` calculates the elastic dispersion diagram – the relation between the acoustic wave number $q$ and frequency $\Omega$– for the problem in the previous tutorial. This is done by scanning over the elastic wavenumber `q_AC` and finding the eigenfrequencies for each value.

As discussed in *Formal selection rules for Brillouin scattering in integrated waveguides and structured fibers* by C. Wolff, M. J. Steel, and C. G. Poulton DOI:/10.1364/OE.22.032489, the elastic modes of any waveguide may be classified according to their representation of the point group symmetry class corresponding to the waveguide profile. For this problem, the waveguide is rectangular with symmetry group $C_{2v}$ which has four symmetry classes, which are marked in the dispersion diagram.

This example also takes advantage of the ability to load and save simulation results to save repeated calculation using the `save_simulation` and `load_simulation`. The previous tutorial saved its electromagnetic results in the file `tut02_wguide_data.npz` using the `Simulation.save_simulation()` method, while the present example recovers those results using `numbat.load_simulation()`. This can be a very useful technique when trying to adjust the appearance of plots without having to repeat the whole calculation effort.

*Note*: from now on, we do not include the code for each tutorial and refer the reader to the relevant files in the `<NumBAT>/examples/tutorials` directory.

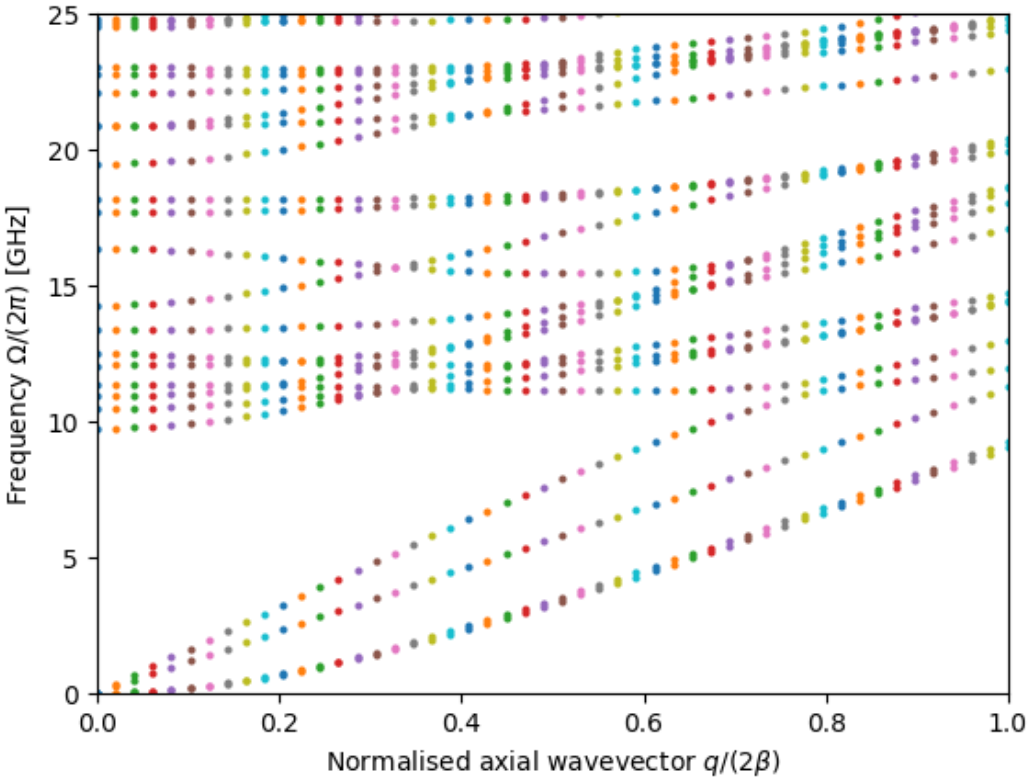



Fig. 8: Acoustic dispersion diagram with modes categorised by symmetry as in Table 1 of Wolff et al. *Opt. Express.* **22**, 32489 (2014).

### 5.2.3 Tutorial 3b – Investigating Dispersion and Multiprocessing

This tutorial, contained in `sim-tut_03_2-dispersion-multicore.py` continues the study of acoustic dispersion and demonstrates the use of Python multiprocessor calls using the `multiprocessing` library to increase speed of execution.

In this code as in the previous example, the acoustic modal problem is repeatedly solved at a range of different $q$ values to build up a set of dispersion curves $\nu_m(q)$. The dispersion diagram looks quite different to the previous case as the waveguide is substantially wider. Due to the large number of avoided and non-avoided crossings, it is usually best to plot dispersion curves like this with dots rather than joined lines. The plot generated below can be improved by increasing the number of $q$ points sampled through the value of the variable `n_qs`, limited only by your patience.

The multiprocessing library runs each task as a completely separate process on the computer. Depending on the nature and number of your CPU, this may improve the performance considerably. This can also be easily extended to multiple node systems which will certainly improve performance. A very similar procedure using the `threading` library allows the different tasks to run as separate threads within the one process. However, due to the existence of the Python Global Interpreter Lock (GIL) which constrains what kinds of operations may run in parallel within Python, multiple threads will typically not improve the performance of NumBAT.

This tutorial also shows an example of saving data, in this case the array of acoustic wavenumbers and frequencies, to a text file using the `numpy` routine `np.savetxt` for later analysis.

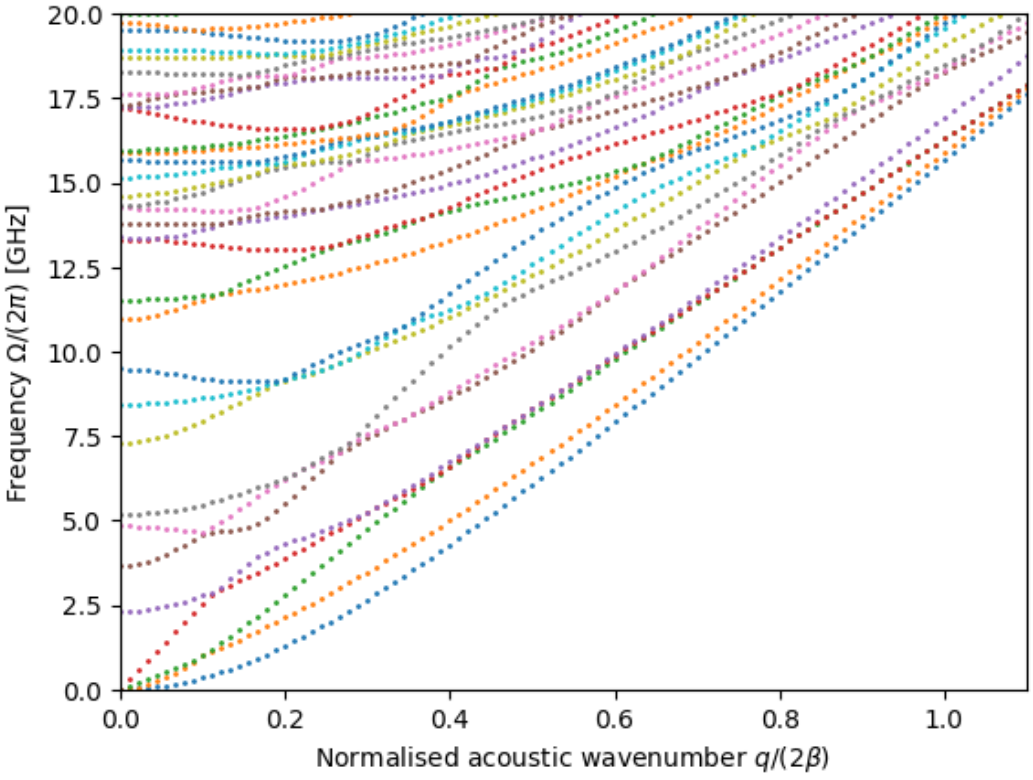



Fig. 9: Acoustic dispersion diagram. The elastic wave number $q$ is scaled by the phase-matched SBS wavenumber $2\beta$ where $\beta$ is the propagation constant of the optical pump mode.

### 5.2.4 Tutorial 4 – Parameter Scan of Widths

This tutorial, contained in `sim-tut_04_scan_widths.py` demonstrates the use of a parameter scan of a waveguide property, in this case the width of the silicon rectangular waveguide, to characterise the behaviour of the Brillouin gain. Later examples in the manual show similar calculations expressed as contour plots rather than in this "waterfall" style.

This calculation generates a great many data files. For this reason, we have provided a second argument to the `NumBATApp` call to specify the name of a new sub-directory, in this case `tut_04-out`, to store all the generated files.

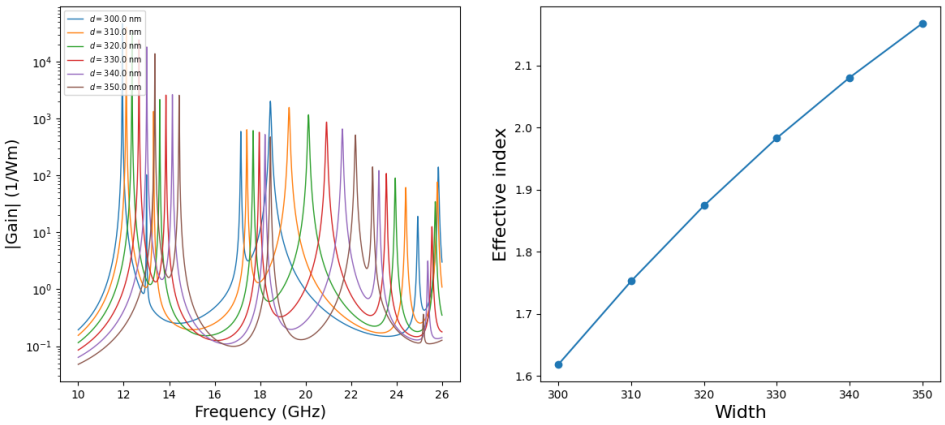



Fig. 10: Gain spectra as function of waveguide width.

### 5.2.5 Tutorial 5 – Convergence Study

This tutorial, contained in `sim-tut_05_convergence_study.py` demonstrates a scan of numerical parameters for our by now familiar silicon-in-air problem to test the convergence of the calculation results. This is done by scanning the value of the `lc_refine` parameters. Since these are two-dimensional FEM calculations, the number of mesh elements (and simulation time) increases with roughly the *square* of the mesh refinement factor.

For the purpose of convergence estimates, the values calculated at the finest mesh (the rightmost values) are taken as the `exact` values, notated with the subscript 0, eg. $\beta_0$. The graphs below show both relative errors and absolute values for each quantity.

Once the convergence properties for a particular problem have been established, it can be useful to do exploratory work more quickly by adopting a somewhat coarser mesh, and then increase the resolution once again towards the end of the project to validate results before reporting them.

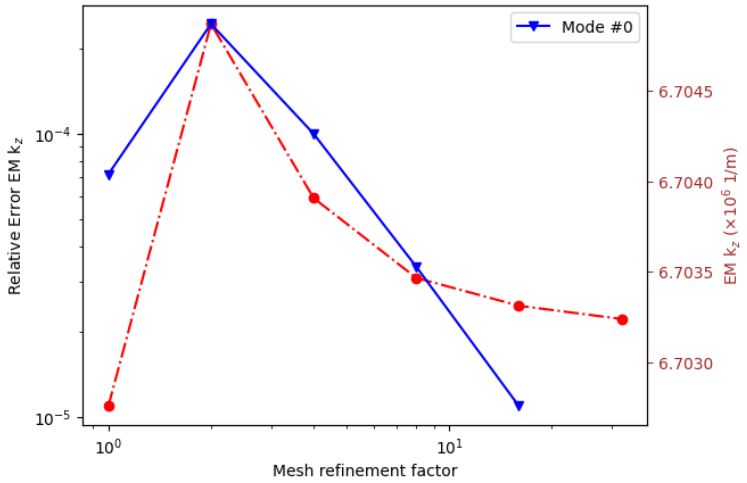



Fig. 11: Convergence of relative (blue) and absolute (red) optical wavenumbers $k_{z,i}$. The left axis displays the relative error $|k_{z,i} - k_{z,0}|/k_{z,0}$. The right axis shows the absolute values of $k_{z,i}$.

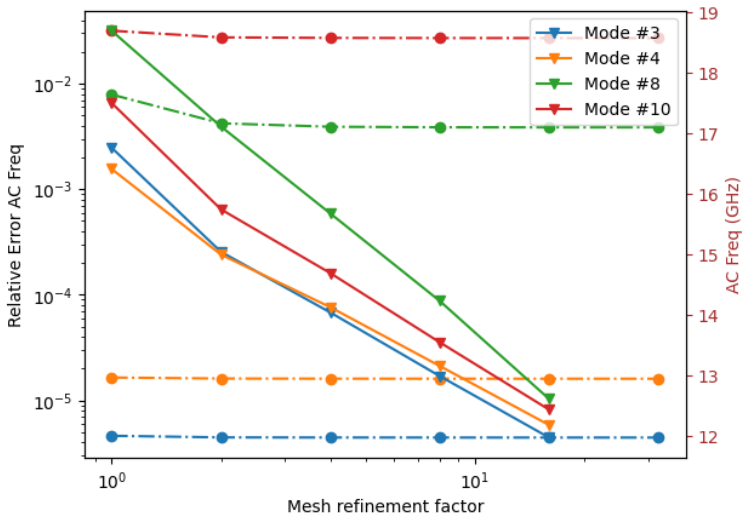



Fig. 12: Convergence of relative (solid, left) and absolute (chain, right) elastic mode frequencies $\nu_i$.

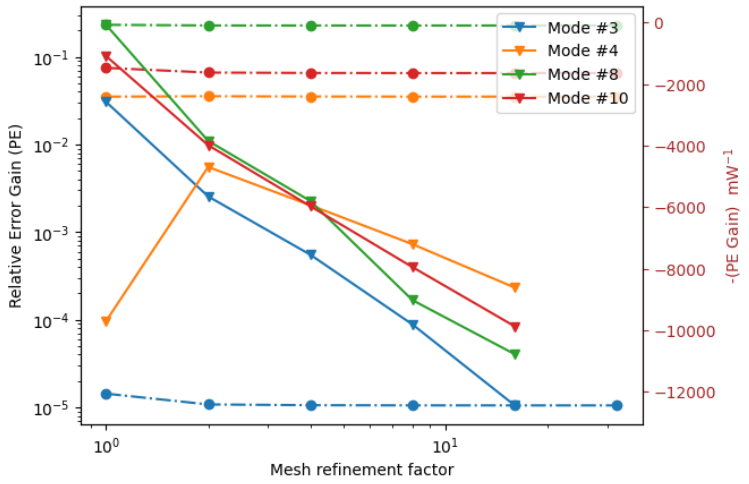



Fig. 13: Convergence of photoelastic gain $G^{\mathrm{PE}}$. The absolute gain on the right hand side increases down the page because of the convention that NumBAT associates backward SBS with negative gain.

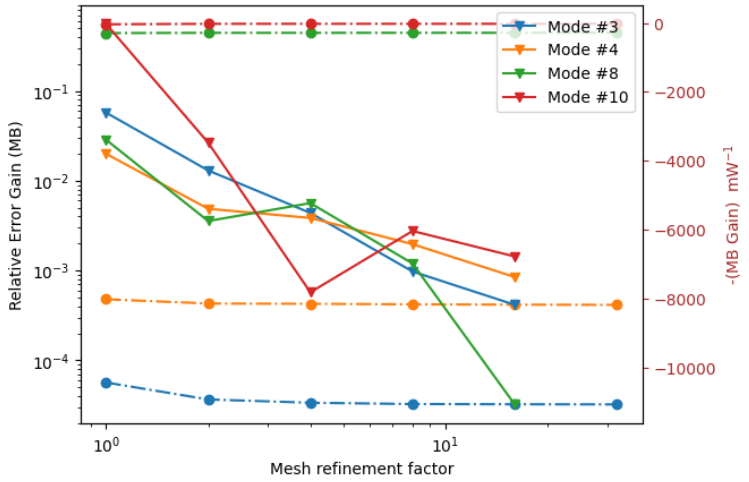



Fig. 14: Absolute and relative convergence of moving boundary gain $G^{\mathrm{MB}}$.

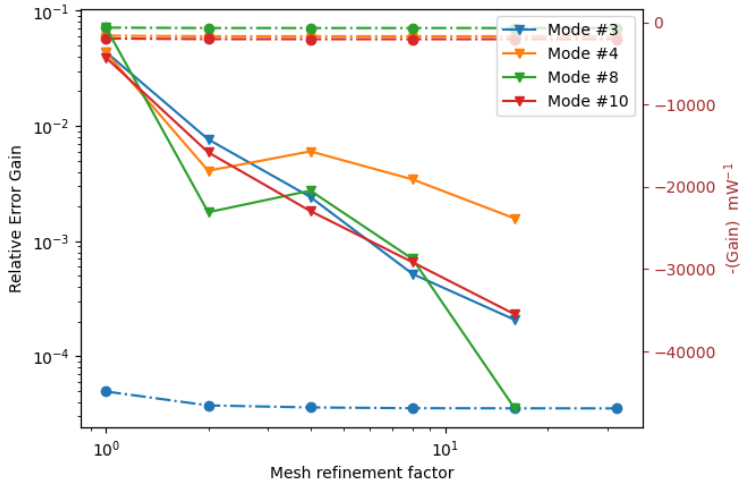



Fig. 15: Absolute and relative convergence of total gain $G$.

### 5.2.6 Tutorial 6 – Silica Nanowire

In this tutorial, contained in `sim-tut_06_silica_nanowire.py` we start to explore the Brillouin gain properties in a range of different structures, in this case a silica circular nanowire surrounded by vacuum.

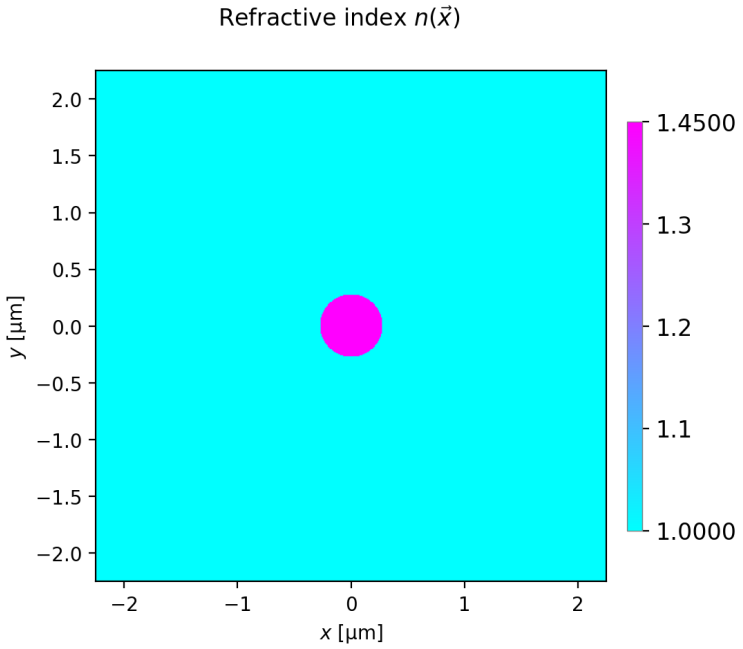



Fig. 16: Refractive index profile of the silica nanowire.

The `gain-spectra` plot below shows the Brillouin gain as a function of Stokes shift. Each resonance peak is marked with the number of the acoustic mode associated with the resonance. This is very helpful in identifying which acoustic mode profiles to examine more closely. In this case, modes 4, , 8 and 23 give the most significant Brillouin gain. The number of modes labelled in the gain spectrum can be controlled using the parameter `mark_mode_threshold` in the function `plot_spectra()` to avoid many labels from modes giving negligible gain. Other parameters allow selecting only one type of gain (PE or MB), changing the frequency range (`freq_min`, `freq_max`), and plotting with log (`logy=True`) or dB (`dB=True`) scales. Note that plots with log scales do not include any noise floor so the peaks look much cleaner than could be observed in the laboratory.

It is important to remember that the total gain is not the simple sum of the photoelastic (PE) and moving boundary (MB) gains. Rather it is the complex coupling amplitudes $Q_{\mathrm{PE}}$ and $Q_{\mathrm{MB}}$ which are added before squaring to give the total gain. Indeed the two effects may have opposite sign so that the net gain can be smaller than either contribution.

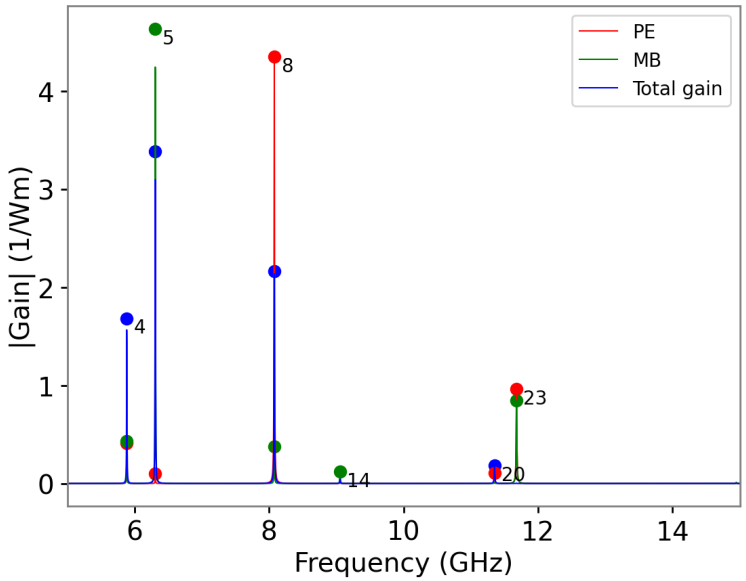



Fig. 17: Gain spectrum showing the gain due to the photoelastic effect (PE), the moving boundary effect (PB), and the net gain (Total).

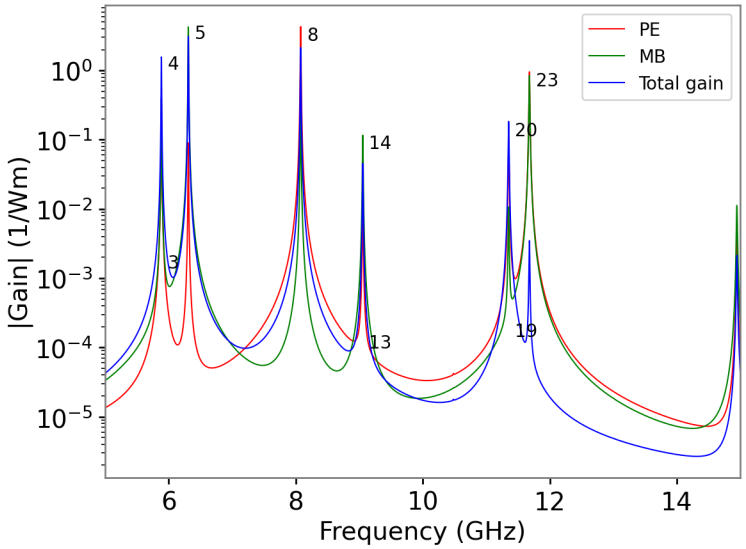



Fig. 18: The same data displayed on a log plot using `logy=True`.

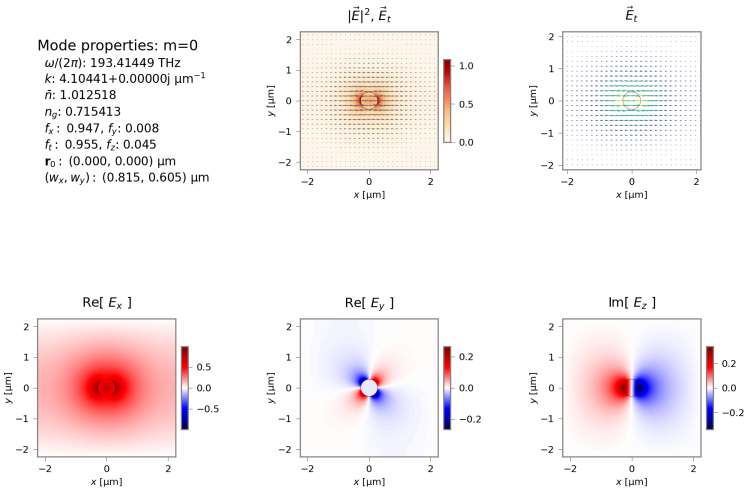



Fig. 19: Electromagnetic mode profile of the pump and Stokes field in the $x$-polarised fundamental mode of the waveguide.

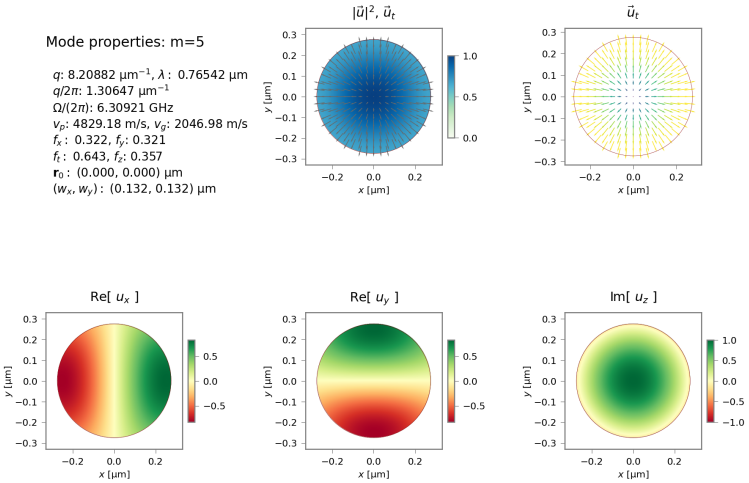



Fig. 20: Mode profiles for acoustic mode 5 which is visible as a MB-dominated peak in the gain spectrum.

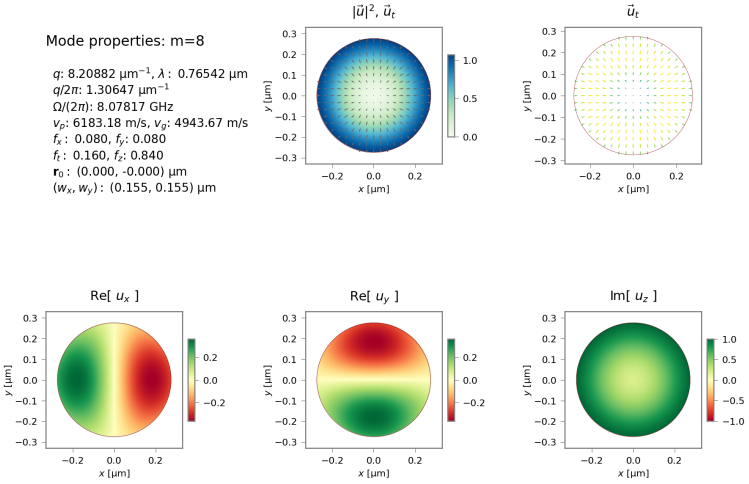



Fig. 21: Mode profiles for acoustic mode 8 which is visible as a PE-dominated peak in the gain spectrum.

### 5.2.7 Tutorial 7 – Slot Waveguide

This tutorial, contained in `sim-tut_07-slot.py` examines backward SBS in a more complex structure: chalcogenide soft glass ($As_2S_3$) embedded in a silicon slot waveguide on a silica slab. This structure takes advantage of the slot effect which expels the optical field into the lower index medium, enhancing the fraction of the EM field inside the soft chalcogenide glass which guides the acoustic mode and increasing the gain.

To understand this, it is helpful to see the refractive index and acoustic velocity profiles. Previously, we have seen how to generate images of the Gmsh template and mesh, but that only gives an indirect sense of the final structure.

In this example, we create structure that can plot the refractive index profile and acoustic velocity profile directly. These are created with the calls `wguide.get_structure_plotter_refractive_index()` and `wguide.get_structure_plotter_acoustic_velocity()`. Then, on each of these structure we can call one or more methods to generate files containing 1D and 2D profiles. The 1D profiles can be made along any x-cut, any y-cut, or along a straight line between any two points.

In the case of the elastic velocity, since there are in general three phase velocities in each material (in this isotropic case, there are two, corresponding to the longitudinal and shear modes), the 1D profiles include all the velocities, and multiple 2D plots are generated.

Here are a few of these.

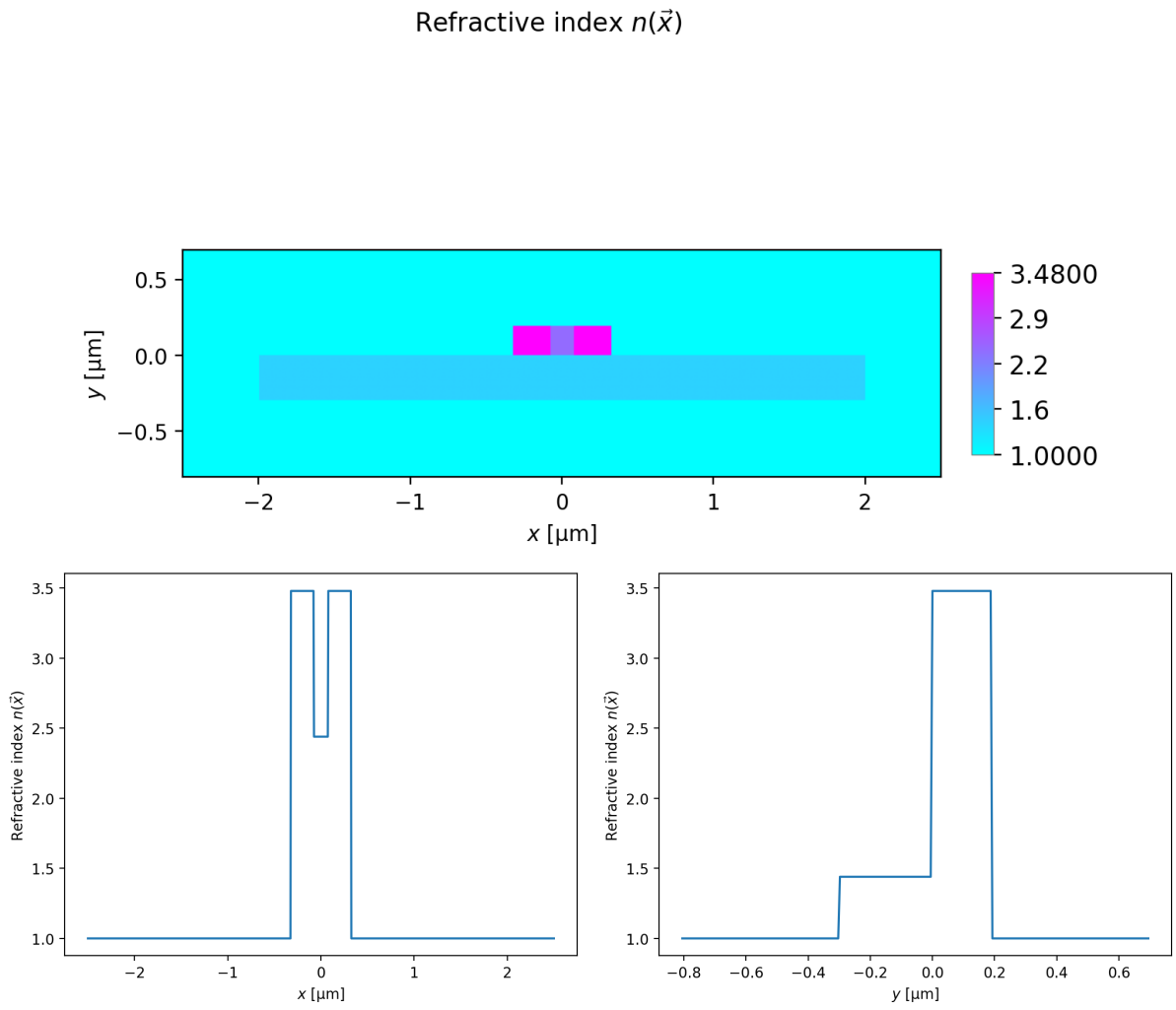



Fig. 22: Refractive index profiles (2D, $x$-cut at $y = 0.1$, $y$-cut at $x = 0.2$ ) of the slot index waveguide.

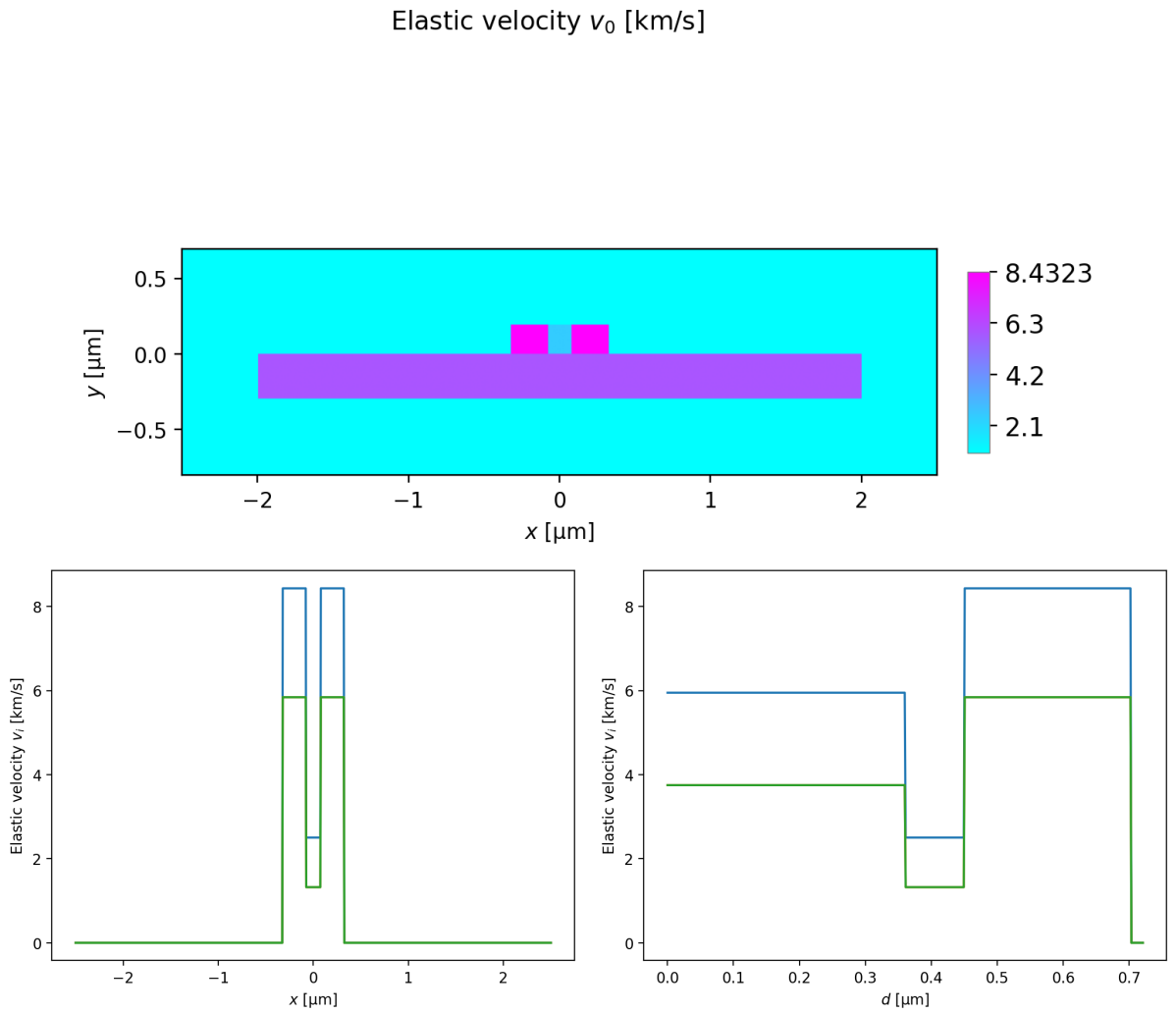



Fig. 23: Elastic velocity profiles (2D, $x$-cut at $y = 0.1$, 1D slice between the points $(-0.3, -0.2)$ and $(0.3, 0.2)$) of the slot index waveguide.

Observe that the refractive index is largest in the pillars surrounding the slot and so the optical localisation to the gap region will be via the *slot effect*. On the other hand, for the elastic problem, *both* the elastic velocities in the gap are lower than in any other part of the structure, and so we can expect one or more elastic modes truly localised to the slot region by total internal reflection.

Now we can look at the gain spectra and mode profiles. The highest gain occurs for elastic modes $m = 0$ and $m = 5$.

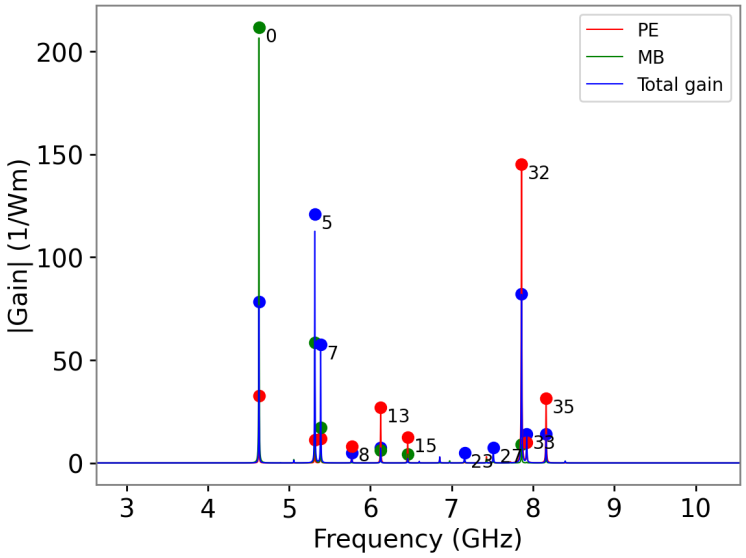



Fig. 24: Gain spectrum showing the gain due to the photoelastic effect (PE), the moving boundary effect (PB), and the net gain (Total).

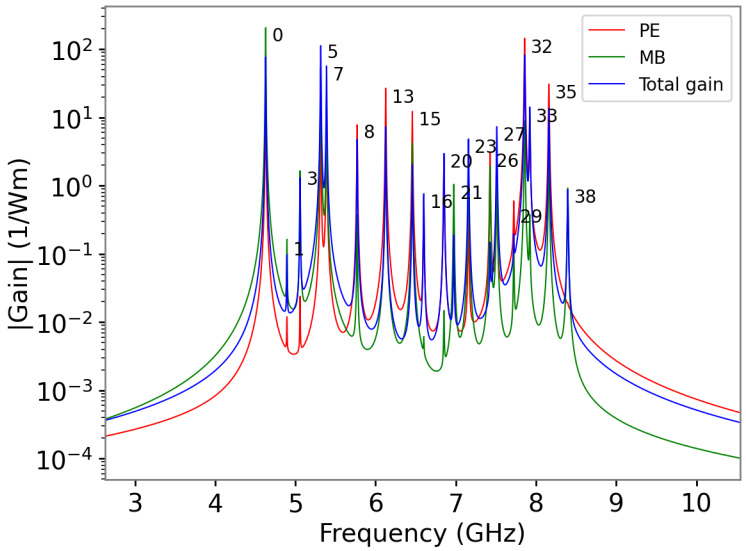



Fig. 25: Gain data shown on a log scale.

Comparing the $m = 0$ and $m = 5$ acoustic mode profiles with the pump EM profile, it is apparent that the field overlap is favourable, whereas the $m = 12$ mode, although being well confined to the slot region, yields zero gain due to its anti-symmetry relative to the pump field.

We also find that the lowest elastic modes are not as localised to the slot region as might be expected. Here, we are seeing a hybridisation of the guided slot mode and Rayleigh-like surface states that are supported on the free boundaries of the slab which is adjacent to the vacuum. This effect could be mitigated by choosing an alternative outer material.

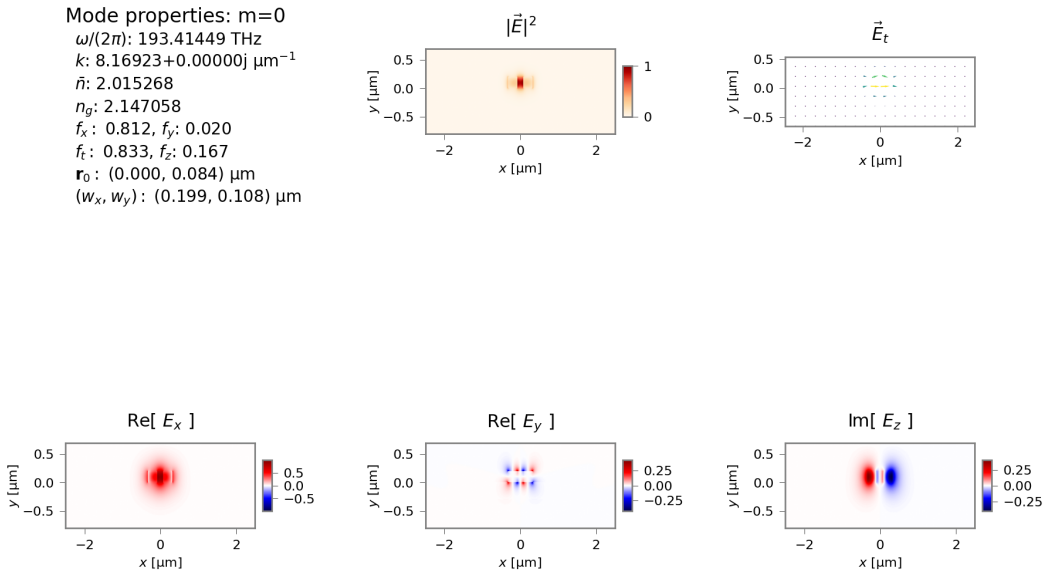



Fig. 26: Electromagnetic mode profile of the pump and Stokes field.

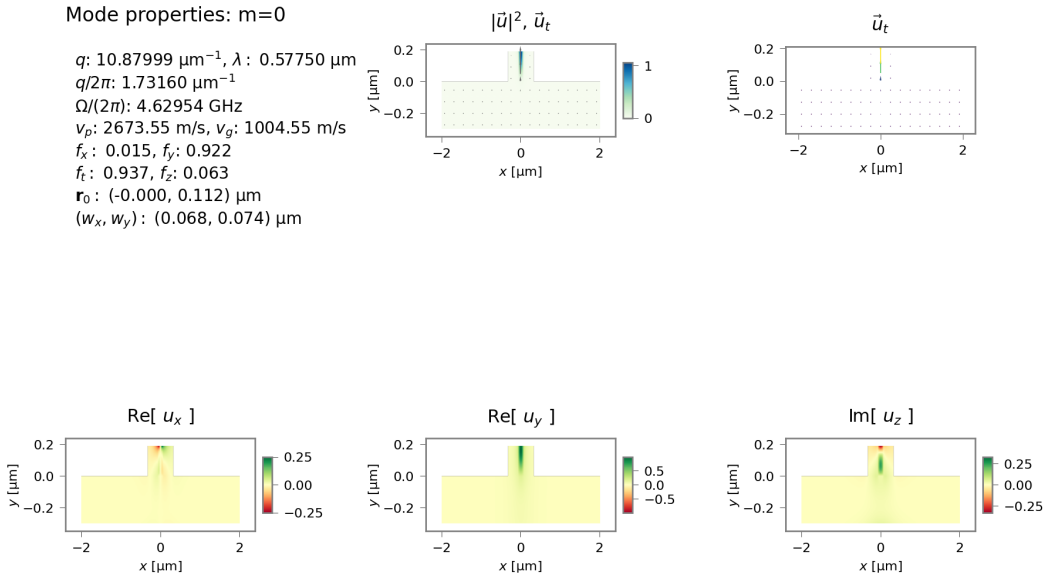



Fig. 27: Acoustic mode profiles for mode 0.

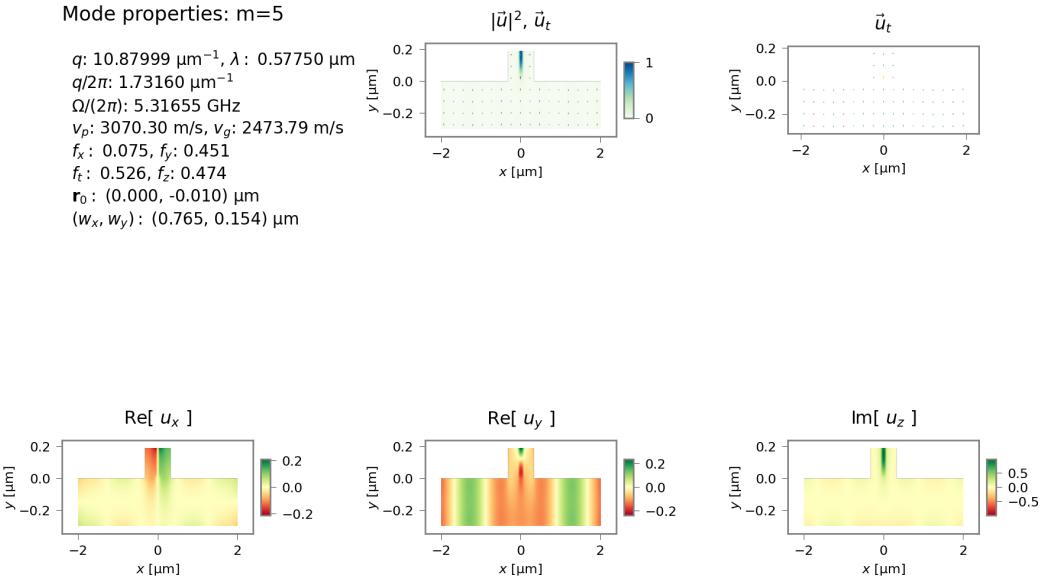



Fig. 28: Acoustic mode profiles for mode 5.

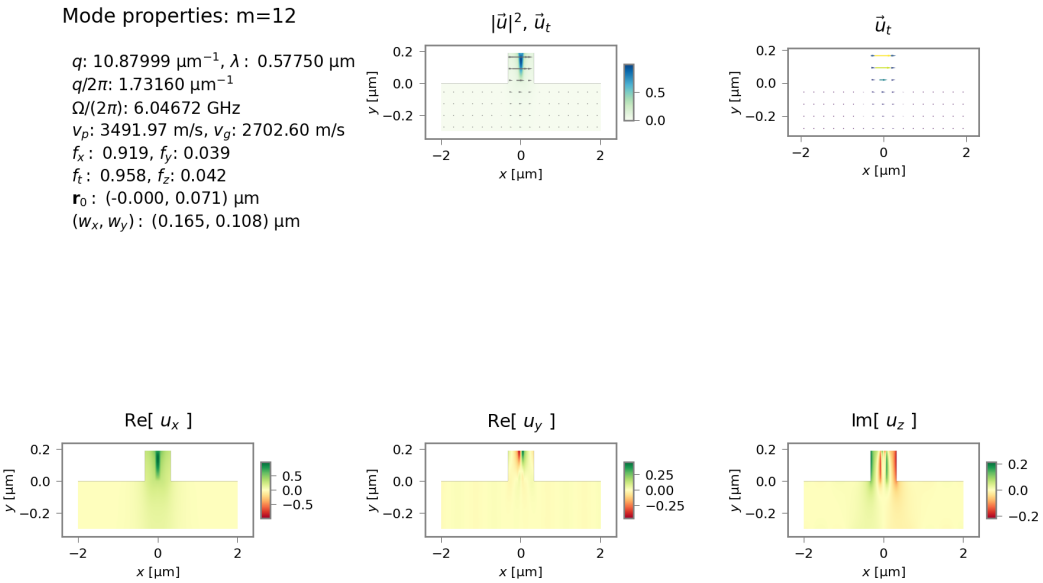



Fig. 29: Acoustic mode profiles for mode 12.

Finally, this simulation file includes examples of plots of 1D cut-profiles along different directions. (Look for the `plot_modes_1D` calls and additional mode outputs in the output fields directory.) Such plots can be useful in resolving features of tightly confined modes.

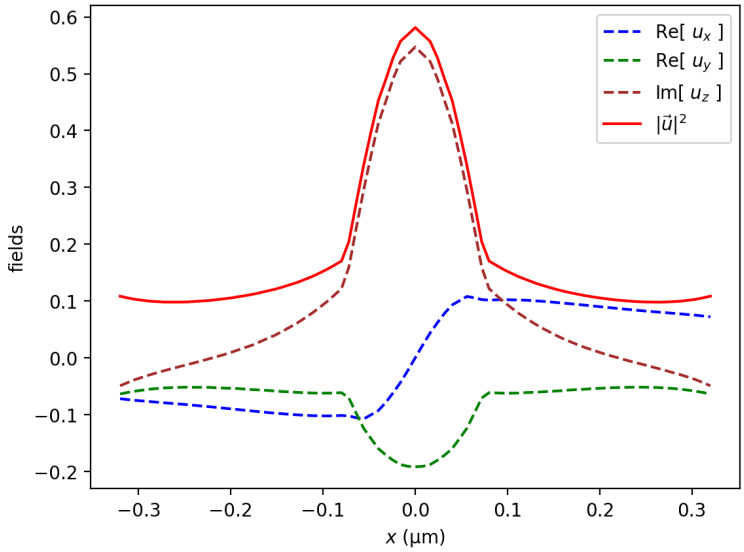



Fig. 30: 1D y-cut mode profiles for mode 5.

### 5.2.8 Tutorial 8 – Slot Waveguide Cover Width Scan

This tutorial, contained in `sim-tut_08-slot_coated-scan.py` continues the study of the previous slot waveguide, by examining the dependence of the acoustic spectrum on the width of the pillars. As before, this parameter scan is accelerated by the use of multi-processing.

The shape of the simulation domain and the compactness of the mode makes the default mode displays hard to see clearly. While this can be addressed with plot settings such as the aspect ratio, and number of vector arrows, it is also helpful to plot each component separately on its own plot. this is achieved with the `comps` option to `plot_modes()`.

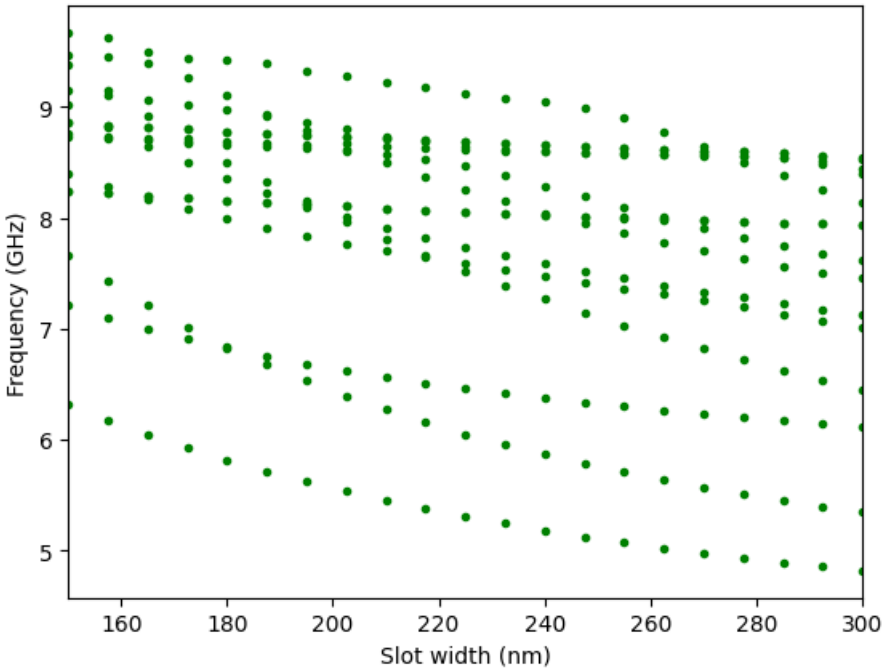



Fig. 31: Acoustic frequencies as function of covering layer thickness.

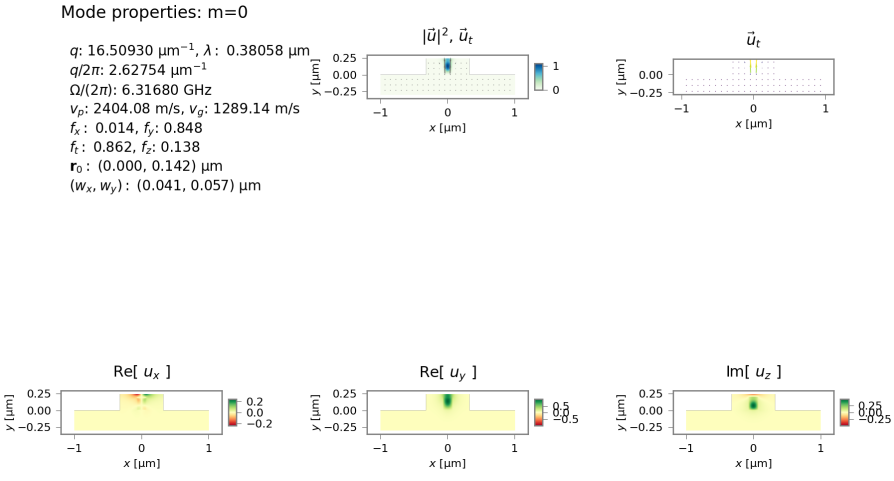



Fig. 32: Modal profiles of lowest acoustic mode.

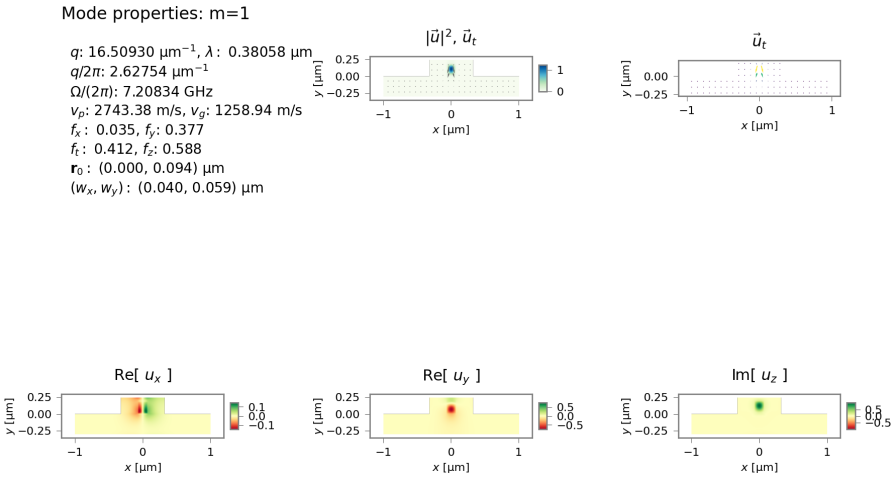



Fig. 33: Modal profiles of second acoustic mode.

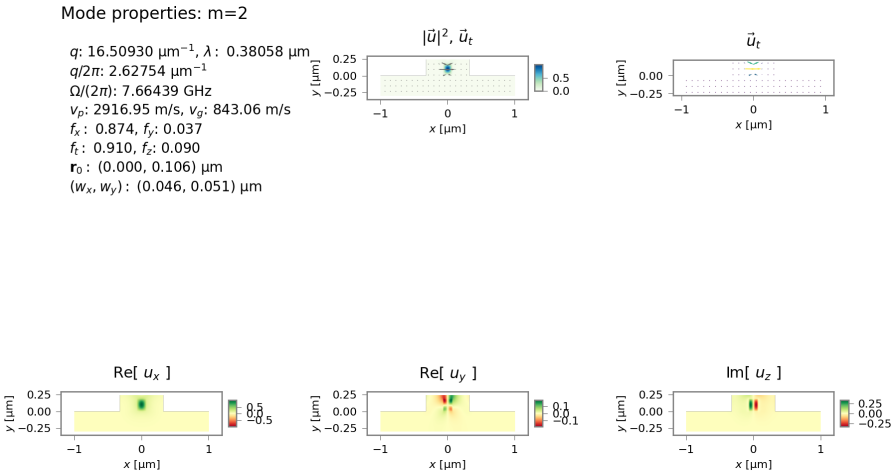
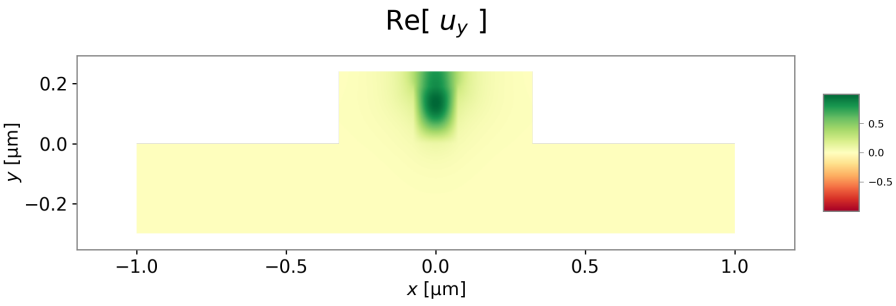



Fig. 34: Modal profiles of third acoustic mode.



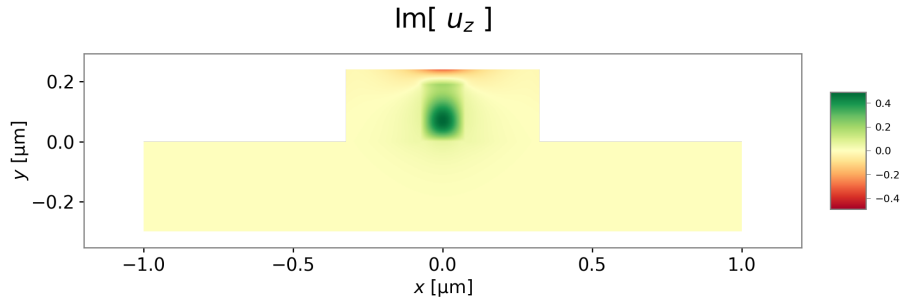



Fig. 35: Individual components of the lowest acoustic mode.

### 5.2.9 Tutorial 9 - Using NumBAT in Jupyter Notebooks

For those who like to work in an interactive fashion, NumBAT works perfectly well inside a Jupyter notebook. This is demonstrated in the file `jup_09_smf28.ipynb` using the standard SMF-28 fibre problem as an example.

On a Linux system, you can open this at the command line with:

```
$ jupyter-notebook ``jup_09_smf28.ipynb``
```

or else load it directly in an already open Jupyter environment.

The notebook demonstrates how to run standard NumBAT routines step by step. The output is still written to disk, so the notebook includes some simple techniques for efficiently displaying mode profiles and spectra inside the notebook.

#### Make some standard inputs.

[1]:
```
%load_ext autoreload
%autoreload 2

import sys
import matplotlib
import matplotlib.pyplot as plt
from IPython.display import Image, display
import glob
import numpy as np
```

[5]:
```
sys.path.append("../backend/")  # or whereever you have NumBATApp installed
import numbat
import materials
import structure
import mode_calcs
import integration
import plotting
#from fortran import numbat
```

#### Specify the geometry

[3]:
```
wl_nm = 1550
domain_x = 5*wl_nm
domain_y = domain_x
inc_a_x = 550
inc_a_y = inc_a_x
inc_shape = 'circular'

num_modes_EM_pump = 20
num_modes_EM_Stokes = num_modes_EM_pump
num_modes_AC = 40
EM_ival_pump = 0
EM_ival_Stokes = 0
AC_ival = 'All'
```

### Make the structure

[ ]:
```
prefix = 'tut_16'
nbapp = numbat.NumBATApp(prefix)

mat_bkg = materials.make_material("Vacuum")
mat_a   = materials.make_material("SiO2_2016_Smith")


wguide = nbapp.make_structure(domain_x,inc_a_x,domain_y,inc_a_y,inc_shape,
                           material_bkg=mat_bkg, material_a=mat_a,
                           lc_bkg=.1, lc_refine_1=10, lc_refine_2=10)
```

```
Building mesh
```

### Calculate the EM modes

[7]:
```
neff_est = 1.4

sim_EM_pump = wguide.calc_EM_modes(num_modes_EM_pump, wl_nm, n_eff=neff_est)
```

```
Calculating EM modes:
 Boundary conditions: Periodic

 Structure has 2089 mesh points and 1024 mesh elements.

 -----------------------------------------------
 EM FEM:
      - assembling linear system for adjoint solution
           cpu time  =   0.17 secs.
           wall time =   0.18 secs.
      - solving linear system
           cpu time  =  17.79 secs.
           wall time =   1.66 secs.
 EM FEM:
      - assembling linear system for prime solution
           cpu time  =   0.17 secs.
           wall time =   0.17 secs.
      - solving linear system
           cpu time  =  15.22 secs.
           wall time =   1.43 secs.
 -----------------------------------------------

  Calculating EM mode powers...
```

Find the backward Stokes fields

[ ]:
```
sim_EM_Stokes = sim_EM_pump.clone_as_backward_modes()
```

### Generate EM mode fields

We are now ready to plot EM field profiles, but how many should we ask for?

The $V$-number of this waveguide can be estimated as $V = \frac{2\pi a}{\lambda}\sqrt{n_c^2 - n_{cl}^2}$:

[ ]:
```
V=2 *pi/wl_nm * inc_a_x * np.sqrt(np.real(mat_a.refindex_n**2
                                  - mat_bkg.refindex_n**2))
print('V={0:.4f}'.format(V))
```

```
V=2.3410
```

We thus expect only a couple of guided modes and to save time and disk space, only ask for the first few to be generated:

[10]:
```
sim_EM_pump.plot_modes(EM_AC='EM_E',
        xlim_min=0.2, xlim_max=0.2, ylim_min=0.2, ylim_max=0.2,
         ivals=range(5))
```

```
Checking triangulation goodness
  Closest space of triangle points was 2.4024988779778966e-08
  No doubled triangles found

  Structure has raw domain(x,y)    = [-3.87500, 3.87500] x [ -3.87500, 3.87500] (um),
                mapped to (x',y') = [-3.87500, 3.87500] x [ -3.87500, 3.87500] (um)

Plotting em modes m=0 to 4.
```

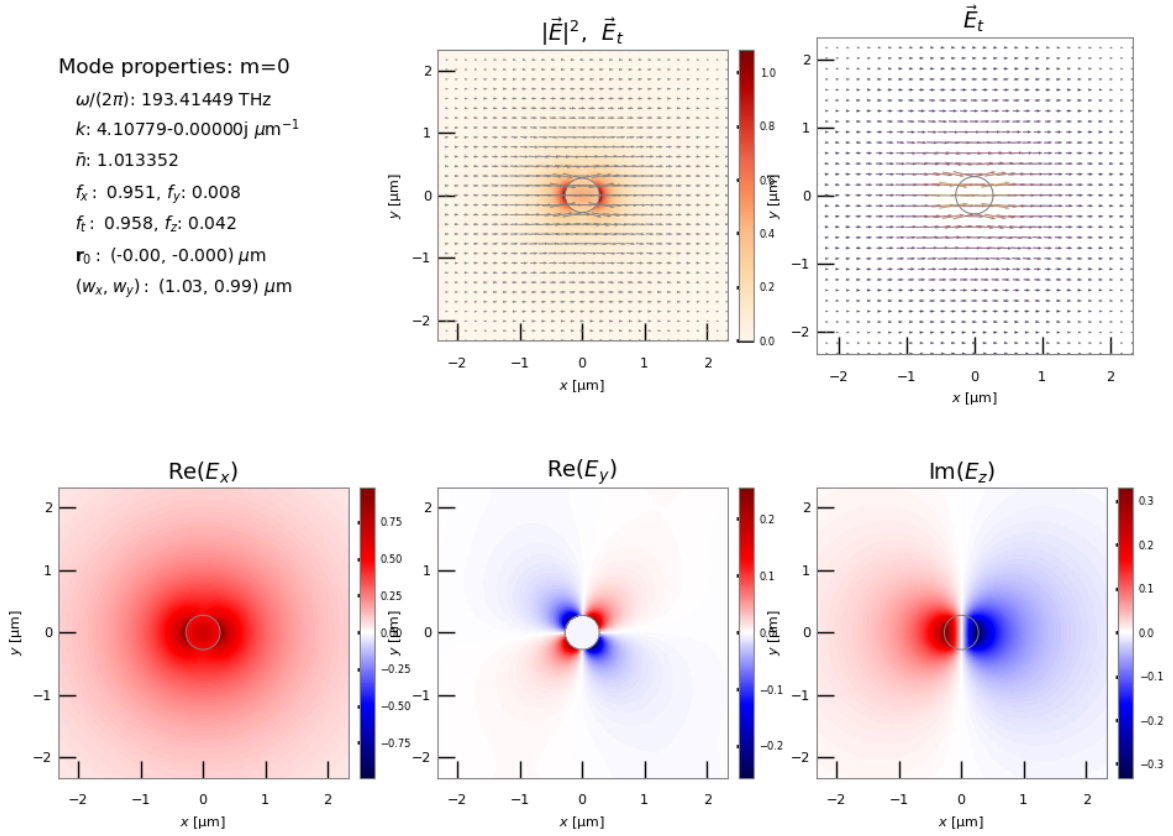
Get a list of the generated files. By sorting the list, the modes will be in order from lowest ($m = 0$) to highest.

[12]:
```
emfields = glob.glob(prefix+'-fields/EM*.png')
emfields.sort()
```

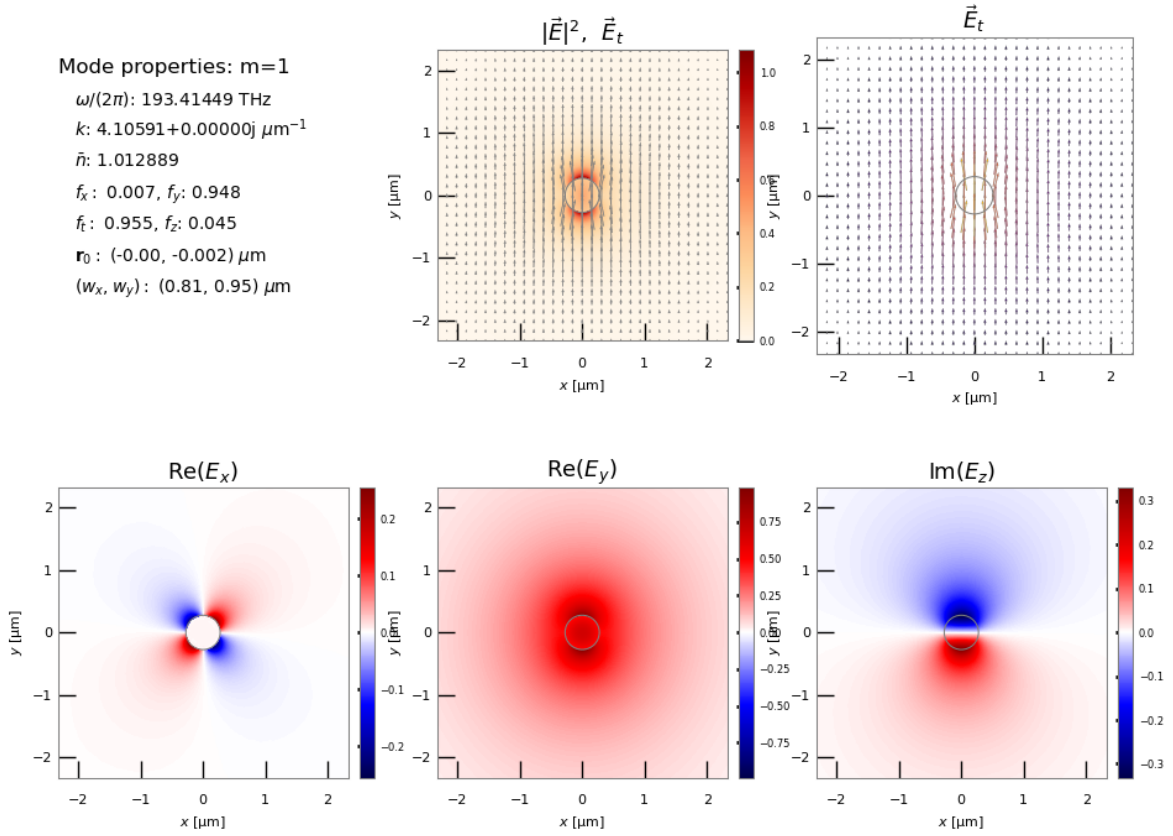
In Jupyter, we can display images using the `display(Image(filename=f))` construct.

[13]:
```
for f in emfields[0:3]:
    print('\n\n',f)
    display(Image(filename=f))
```

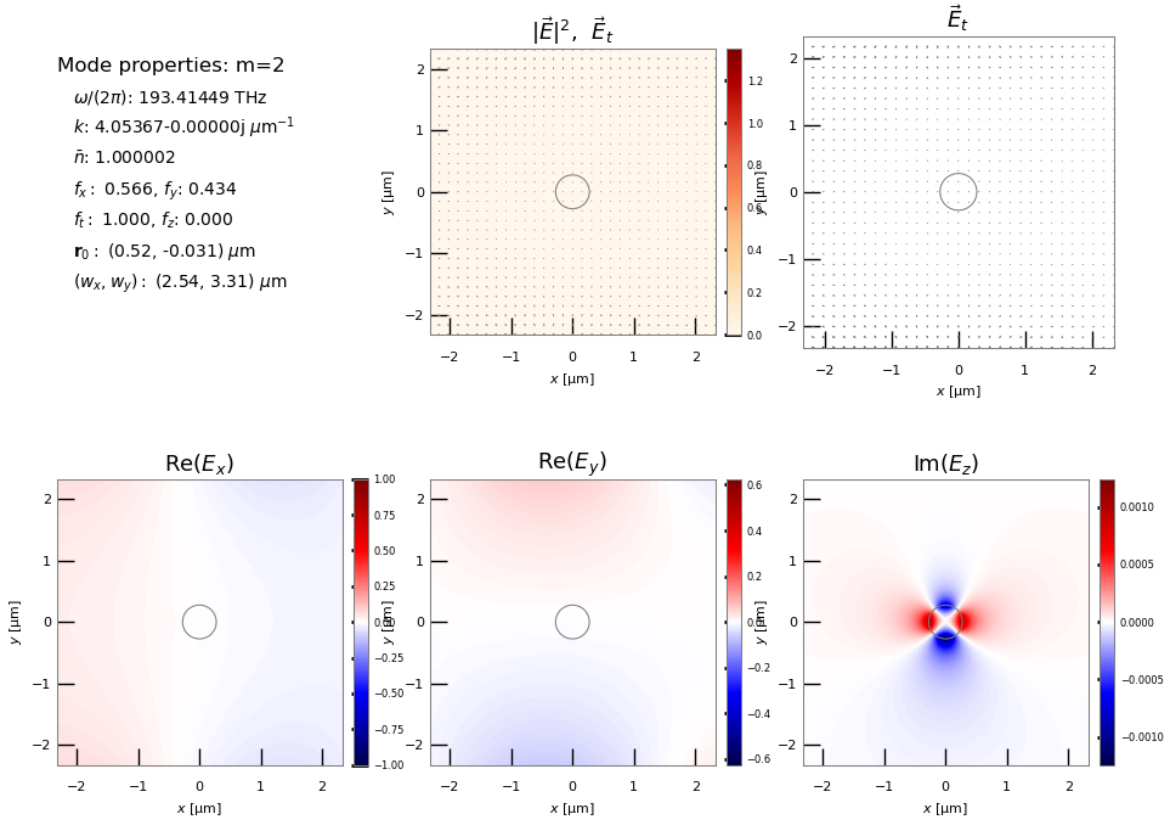
```


 tut_16-fields/EM_E_mode_00.png
```




tut_16-fields/EM_E_mode_01.png




tut_16-fields/EM_E_mode_02.png




## Calculate the acoustic modes

Now let's turn to the acoustic modes.

For backwards SBS, we set the desired acoustic wavenumber to the difference between the pump and Stokes wavenumbers. Ω We specify a 'shift' frequency as a starting location of the frequency to look for solutions

[14]:
```
q_AC = np.real(sim_EM_pump.kz_EM(EM_ival_pump) - sim_EM_Stokes.kz_EM(EM_ival_Stokes))

NuShift_Hz = 4e9

sim_AC = wguide.calc_AC_modes(num_modes_AC, q_AC, EM_sim=sim_EM_pump, shift_
↪Hz=NuShift_Hz)
```

```
Calculating AC modes

 Structure has 273 mesh points and 124 mesh elements.

 -------------------------------------------------
 AC FEM:
       - assembling linear system
           cpu time  =   0.00 secs.
           wall time =   0.00 secs.
       - solving linear system
           cpu time  =   0.77 secs.
           wall time =   0.05 secs.
 -------------------------------------------------
```

(continues on next page)

(continued from previous page)

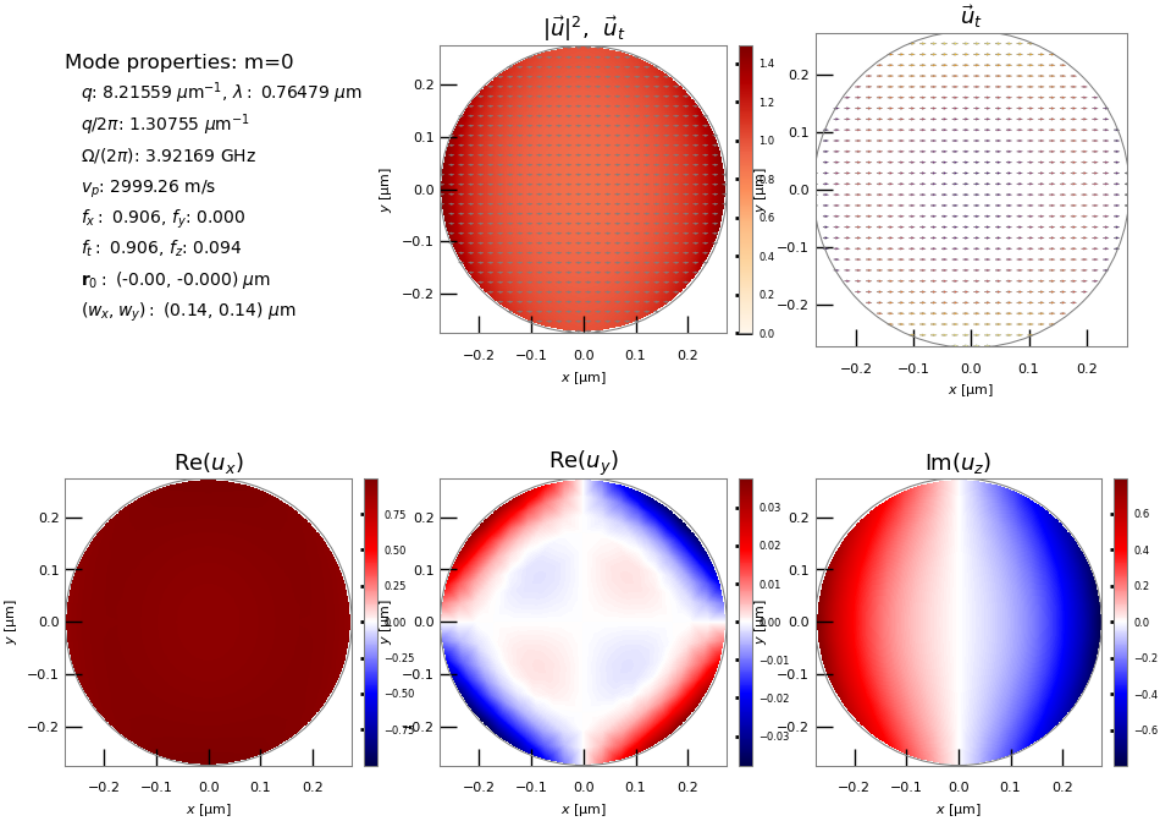
[15]:
```
sim_AC.plot_modes( ivals=range(10))
```

```
Checking triangulation goodness
  Closest space of triangle points was 2.4024988779778966e-08
  No doubled triangles found

  Structure has raw domain(x,y)    = [-0.27500, 0.27500] x [ -0.27500, 0.27500] (um),
                mapped to (x',y') = [-0.27500, 0.27500] x [ -0.27500, 0.27500] (um)

Plotting acoustic modes m=0 to 9.
```

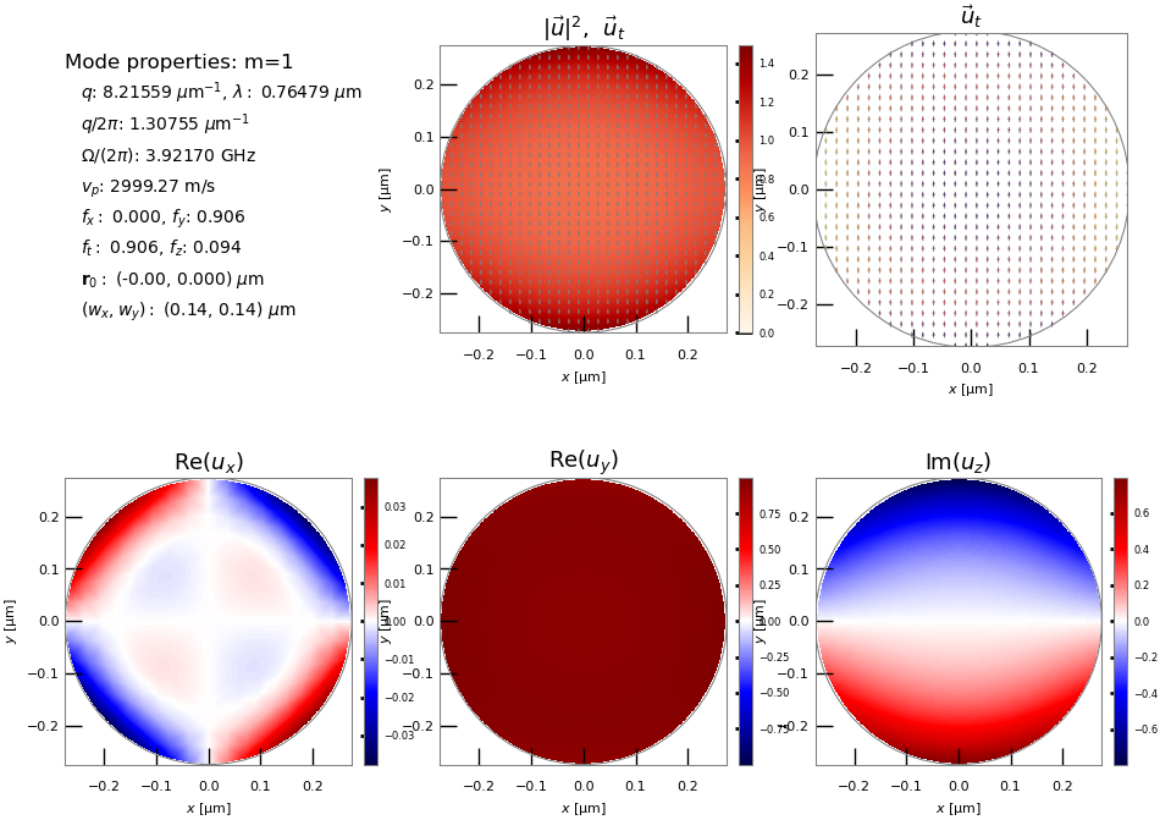
[16]:
```
acfields = glob.glob(prefix+'-fields/AC*.png')
acfields.sort()
```

[17]:
```
for f in acfields[0:6]:
    print('\n\n',f)
    display(Image(filename=f))
```

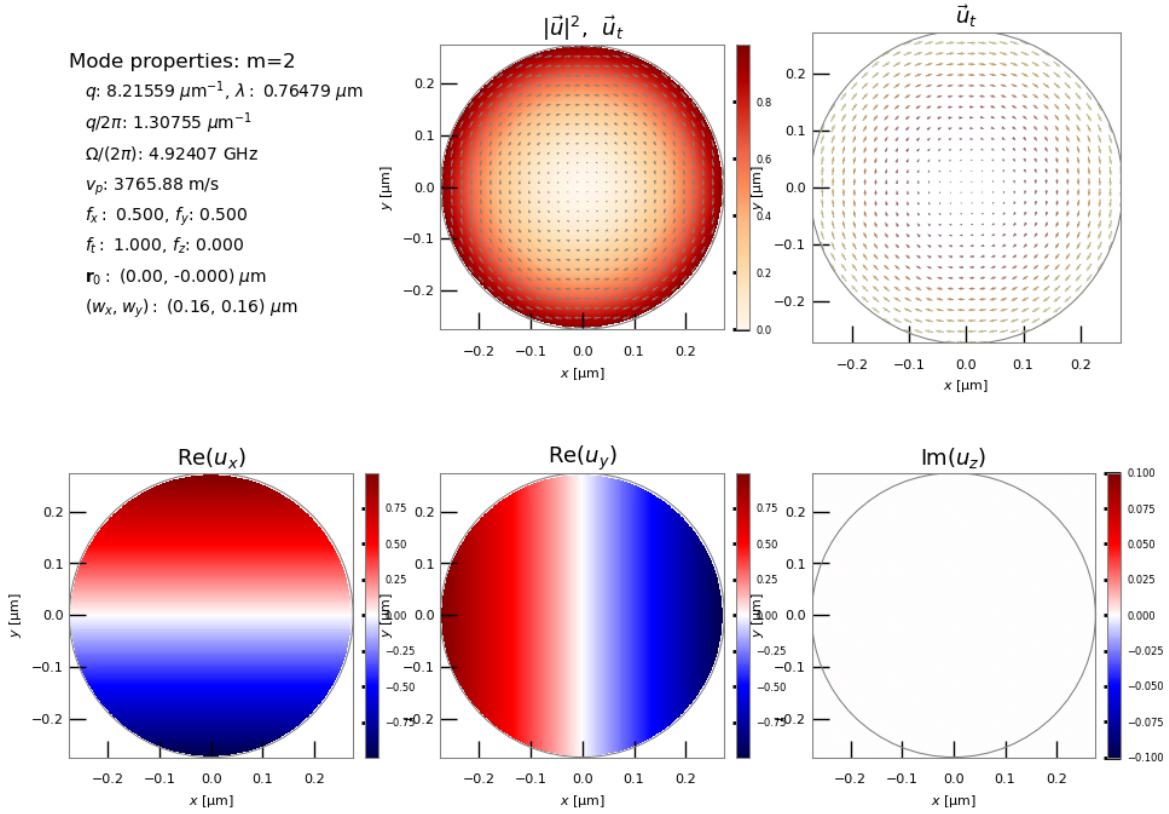
```
 tut_16-fields/AC_mode_00.png
```



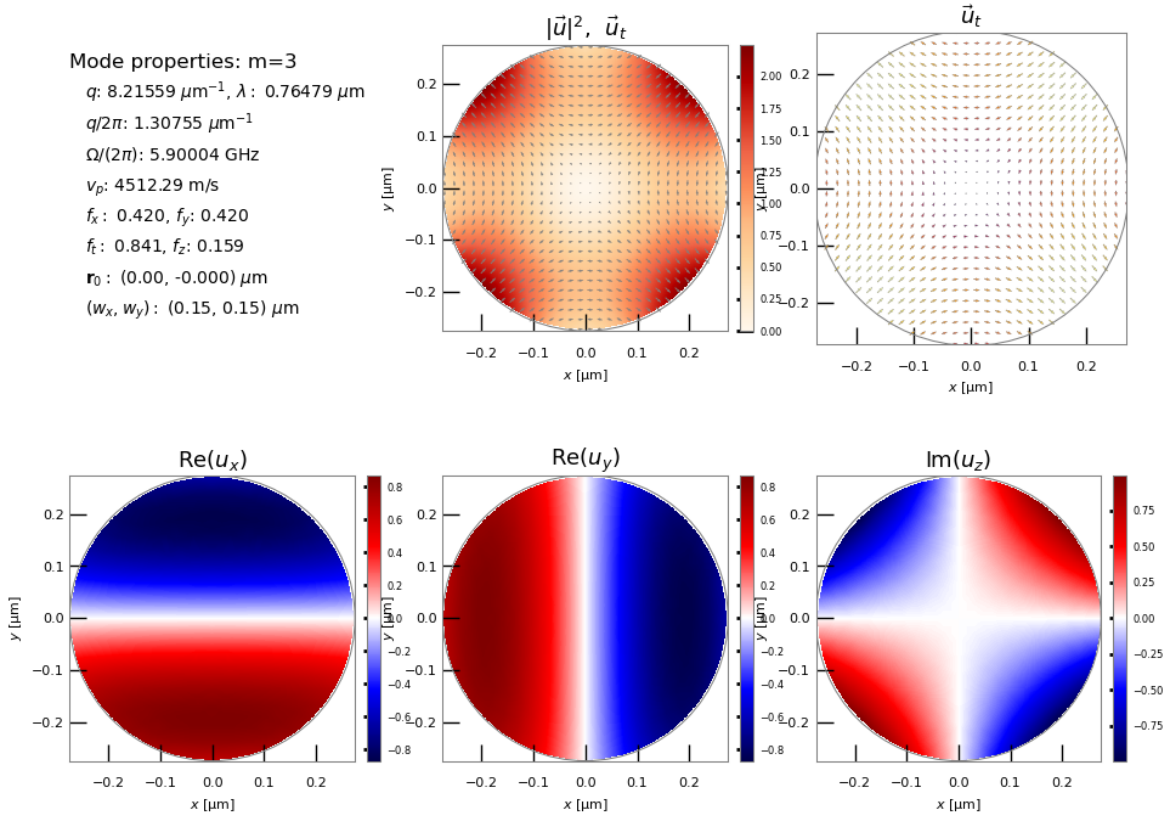

```
 tut_16-fields/AC_mode_01.png
```



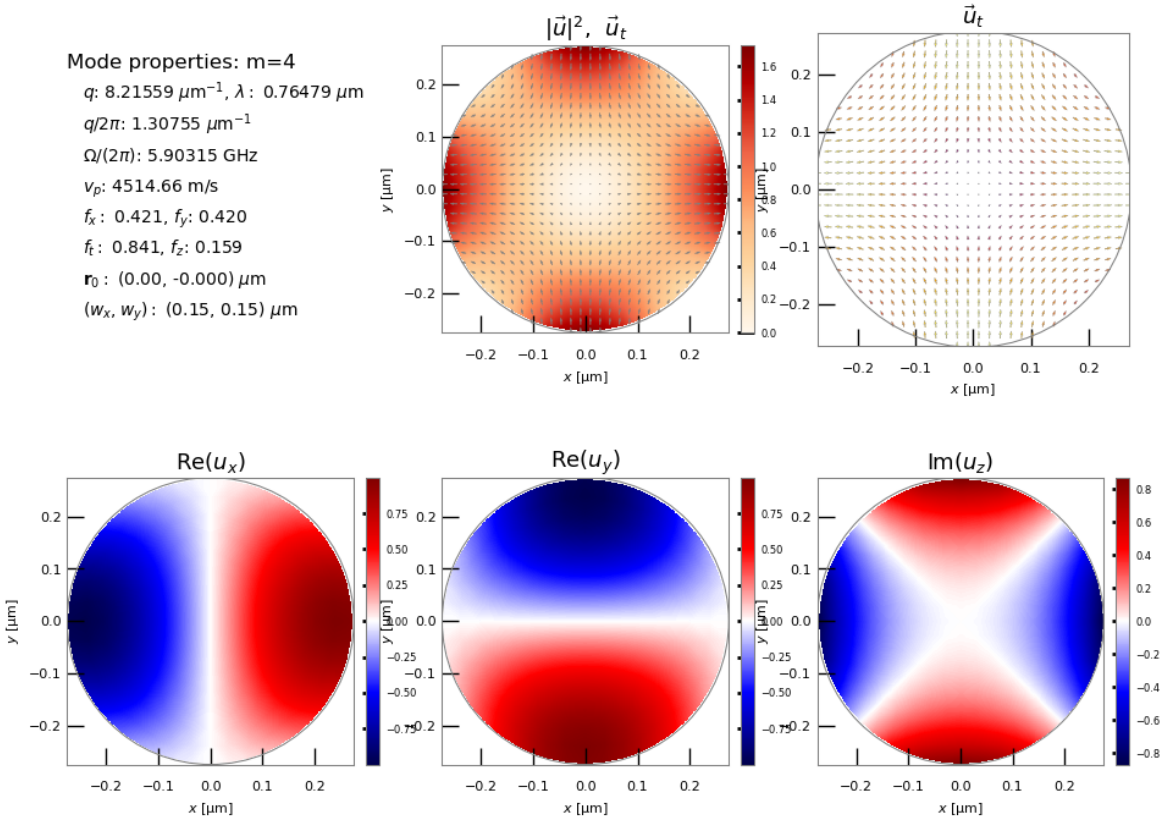

tut_16-fields/AC_mode_02.png



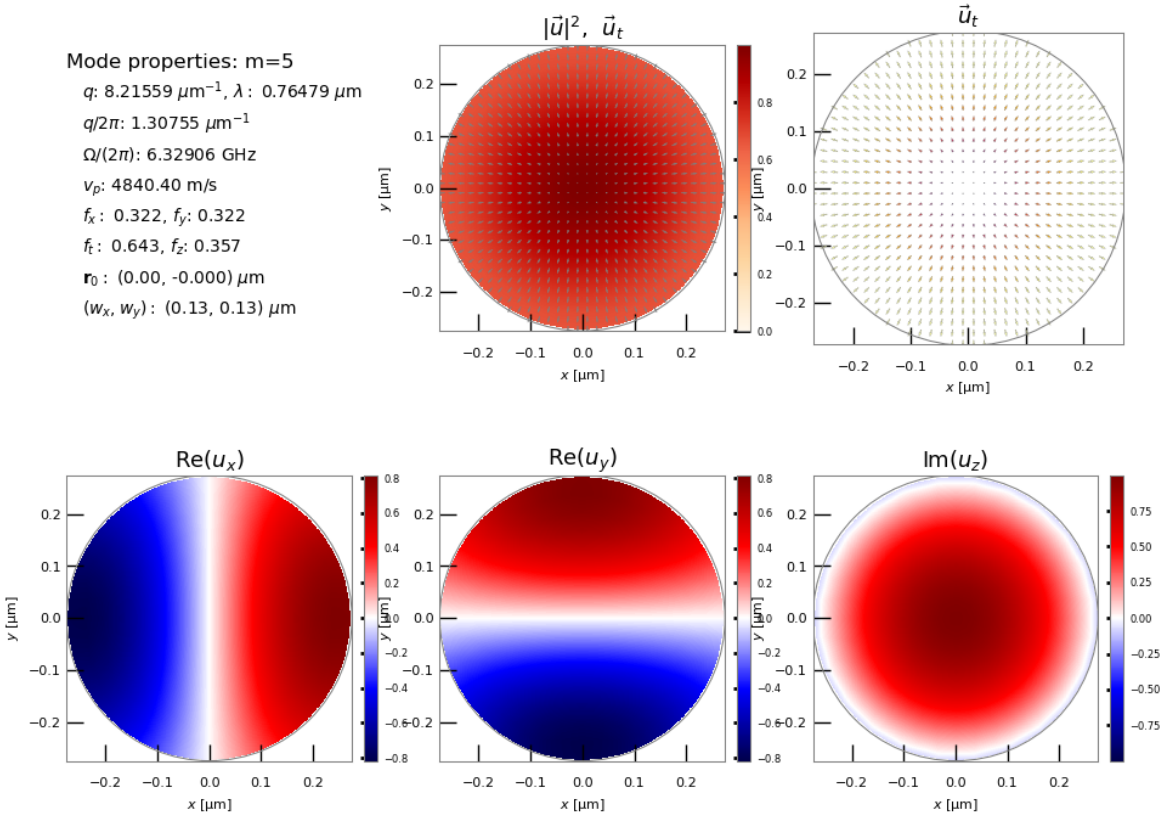

tut_16-fields/AC_mode_03.png




tut_16-fields/AC_mode_04.png




tut_16-fields/AC_mode_05.png


Mode properties: m=5
q: 8.21559 μm^−1, λ : 0.76479 μm
q/2π: 1.30755 μm^−1
Ω/(2π): 6.32906 GHz
v_p: 4840.40 m/s
f_x : 0.322, f_y: 0.322
f_t : 0.643, f_z: 0.357
r_0 : (0.00, -0.000) μm
(w_x, w_y) : (0.13, 0.13) μm
|u⃗|², u⃗_t
u⃗_t
Re(u_x)
Re(u_y)
Im(u_z)
x [μm]
y [μm]

[ ]:

CHAPTER

SIX

# MATERIALS, WAVEGUIDES AND MESHING

Now that we have a basic understanding of using NumBAT, this chapter provides detailed information on how to specify a large of range materials and waveguide designs.

We will return to more advanced examples/tutorials in the next chapter.

## 6.1 Materials

In order to calculate the modes of a structure we must specify the acoustic and optical properties of all constituent materials.

In NumBAT, this data is read in from human-readable `.json` files, which are stored in the directory `<NumBAT>/backend/material_data`.

These files not only provide the numerical values for optical and acoustic variables, but provide links to the origin of the data. Often they are taken from the literature and the naming convention allows users to select from different parameter values chosen by different authors for the same nominal material.

The intention of this arrangement is to create a library of materials that can serves as standard reference data within the research community. They also allow users to check the sensitivity of their results on particular parameters for a given material.

**At present, the library contains the following materials:**

- **Vacuum (or air)**
  - `Vacuum`
- **The chalcogenide glass Arsenic tri*sulfide**
  - `As2S3_2016_Smith`
  - `As2S3_2017_Morrison`
  - `As2S3_2021_Poulton`
- **Fused silica**
  - `SiO2_2013_Laude`
  - `SiO2_2015_Van_Laer`
  - `SiO2_2016_Smith`
  - `SiO2_2021_Smith`
  - `SiO2_smf28.json`
  - `SiO2GeO2_smf28.json`
- **Silicon**
  - `Si_2012_Rakich`
  - `Si_2013_Laude`

  - Si_2015_Van_Laer
  - Si_2016_Smith
  - Si_2021_Poulton
  - Si_test_anisotropic
- **Silicon nitride**
  - Si3N4_2014_Wolff
  - Si3N4_2021_Steel
- **Gallium arsenide**
  - GaAs_2016_Smith
- **Germanium**
  - Ge_cubic_2014_Wolff
- **Lithium niobate**
  - LiNbO3_2021_Steel
  - LiNbO3aniso_2021_Steel

Materials can easily be added to this library by copying any of these files as a template and modifying the properties to suit. The `Si_test_anisotropic` file contains all the variables that NumBAT is setup to read. We ask that stable parameters (particularly those used for published results) be added to the NumBAT git repository using the same naming convention.

## 6.2 Waveguide Geometries

NumBAT encodes different waveguide structures through finite element meshes constructed using the `.geo` language used by the open source tool `Gmsh`. Most users will find they can construct all waveguides of interest using the existing templates. However, new templates can be added by adding a new `.geo` file to the `<NumBAT>/backend/fortran/msh` directory and making a new subclass of the `UserGeometryBase` class in the `<NumBAT>/backend/msh/user_meshes.py` file. This procedure is described in detail in *User-defined waveguide geometries*.

All the builtin examples below are constructed in the same fashion in a parallel `builtin_meshes.py` file and can be used as models for your own designs.

The following figures give some examples of how material types and physical dimensions are represented in the mesh geometries. In particular, for each structure template, they identify the interpretation of the dimensional parameters (`inc_a_x`, `slab_b_y`, etc), material labels (`material_a`, `material_b` etc), and the grid refinement parameters (`lc_bkg`, `lc_refine_1`, `lc_refine_2`, etc). The captions for each structure also identify the mesh geometry template files in the directory `<NumBAT>/backend/fortran/msh` with filenames of the form `<prefix>_msh_template.geo` which define the structures and can give ideas for developing new structure files.

The NumBAT code for creating all these structures can be found in `<NumBAT>/docs/source/images/make_meshfigs.py`.

### 6.2.1 Single inclusion waveguides with surrounding medium

These structures consist of a single medium inclusion (`mat_a`) with a background material (`mat_bkg`). The dimensions are set with `inc_a_x` and `inc_a_y`.

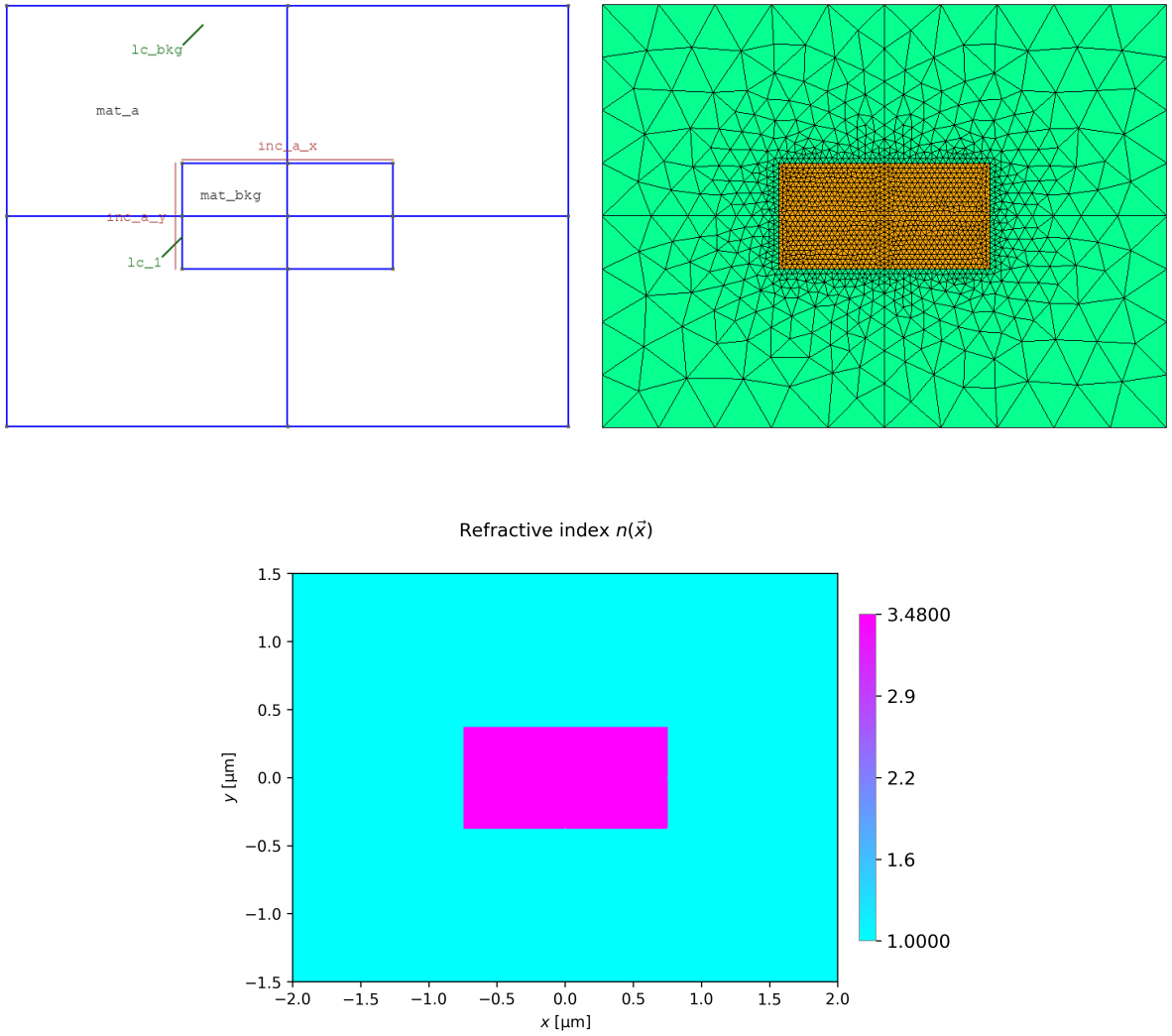
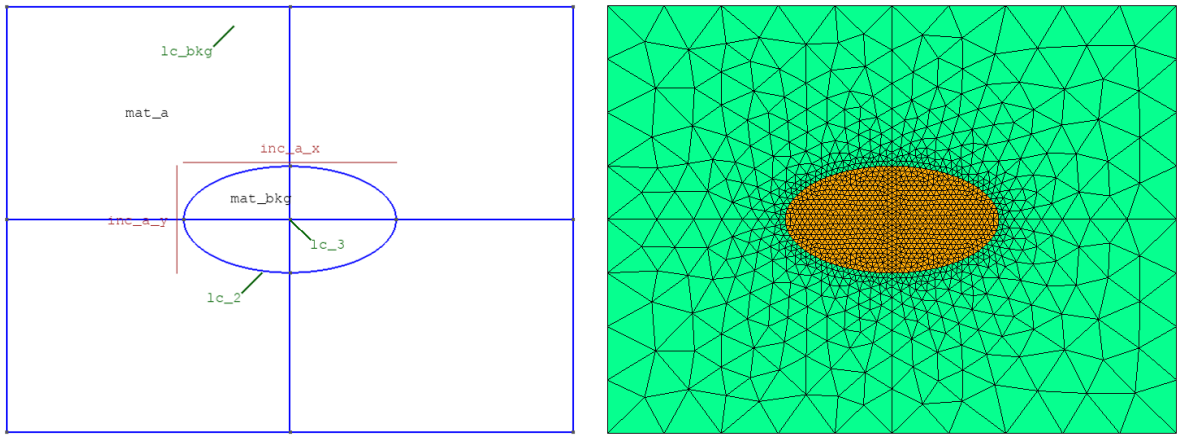



Fig. 1: Rectangular waveguide using shape `rectangular` (template `oneincl_msh`).



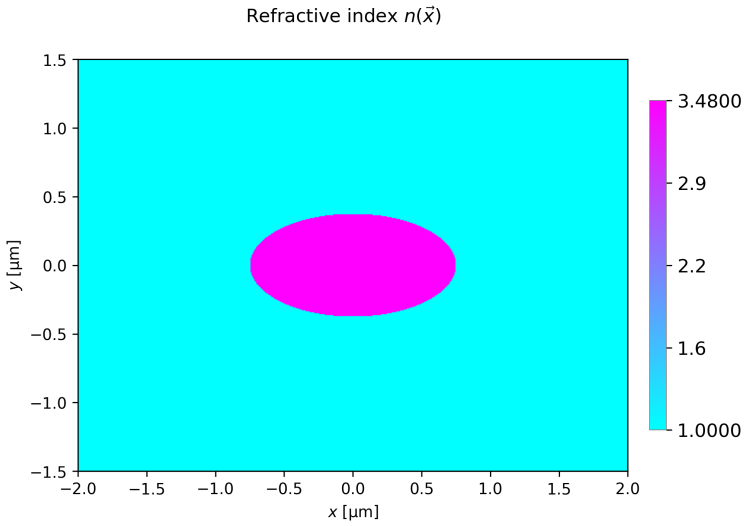
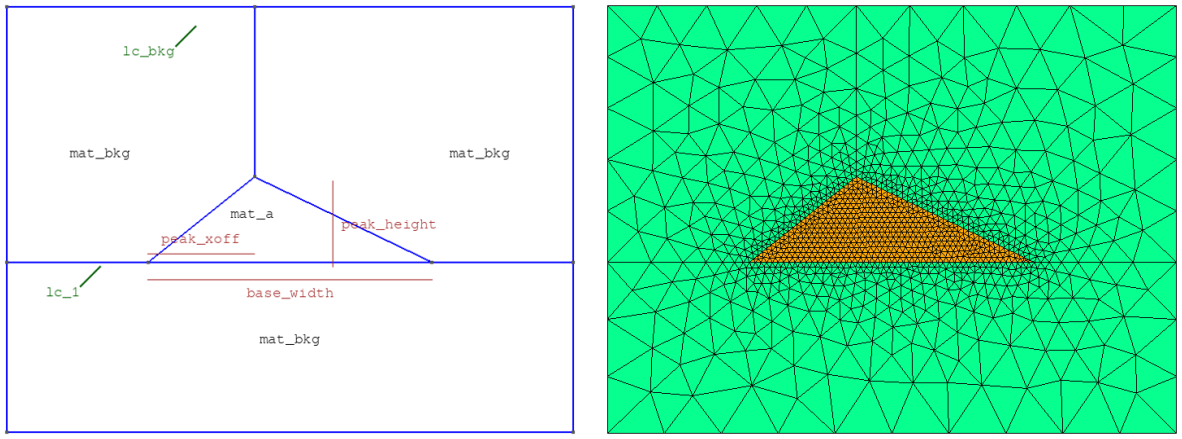



Fig. 2: Elliptical waveguide using shape `circular` (template `oneincl_msh`).



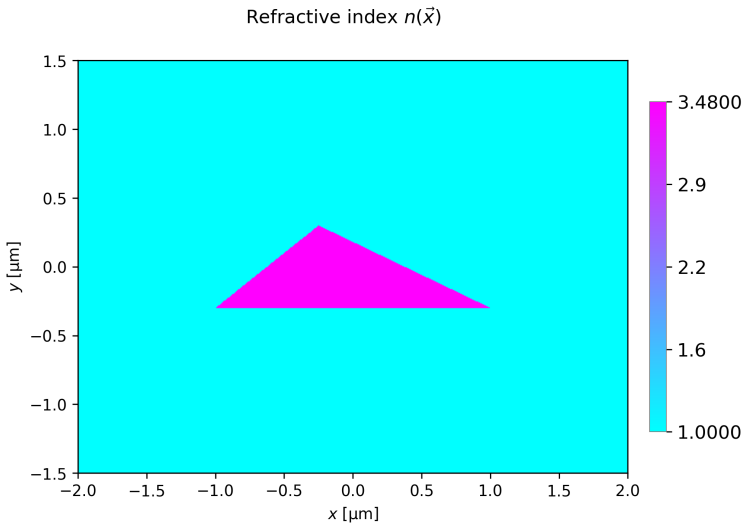
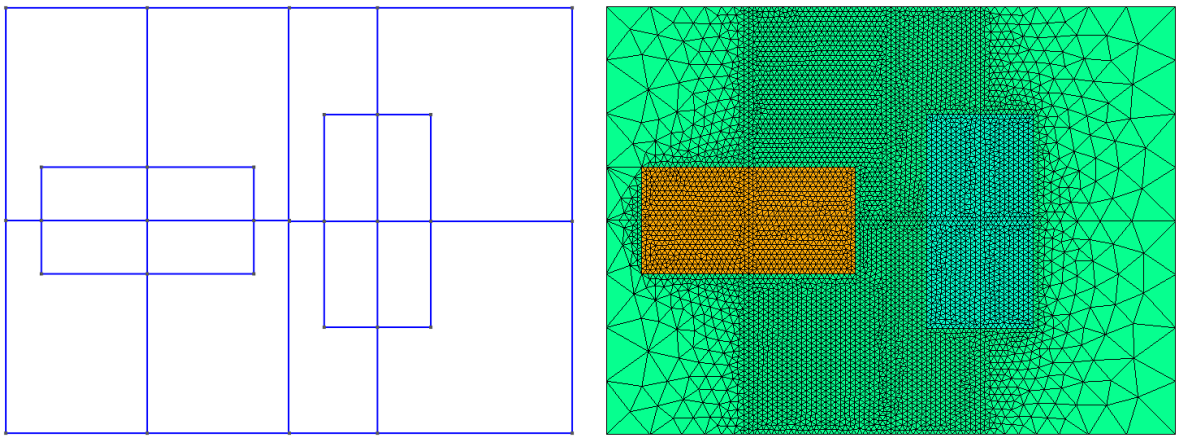




Fig. 3: Triangular waveguide using shape `triangular`.

### 6.2.2 Double inclusion waveguides with surrounding medium

These structures consist of a pair of waveguides with a single common background material. The dimensions are set by `inc_a_x/inc_a_y` and `inc_b_x/inc_b_y`. They are separated horizontally by `two_inc_sep` and the right waveguide has a vertical offset of `y_off`.



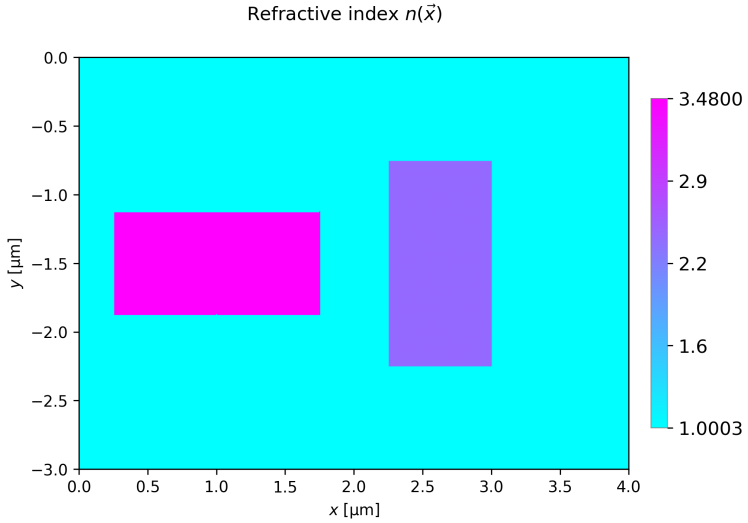
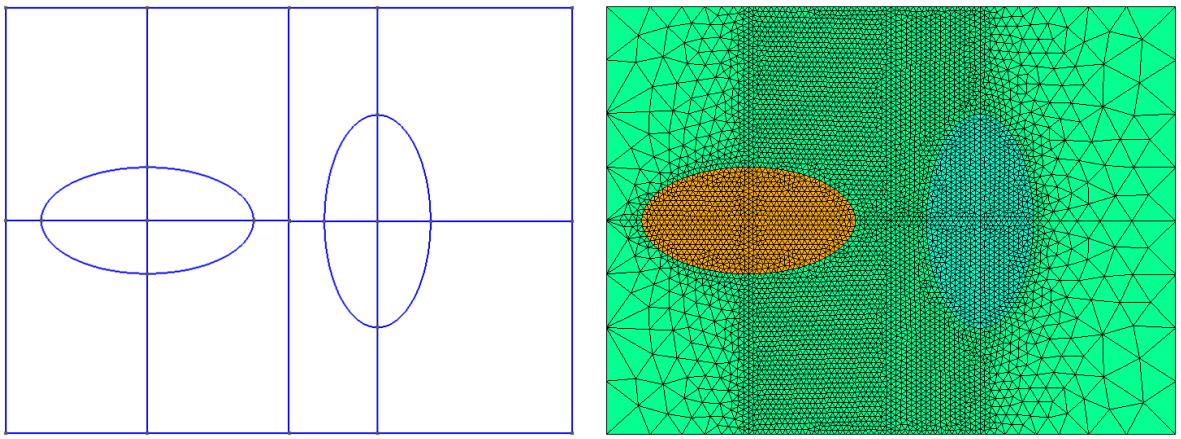



Fig. 4: Coupled rectangular waveguides using shape `rectangular` (template `twoincl_msh`).



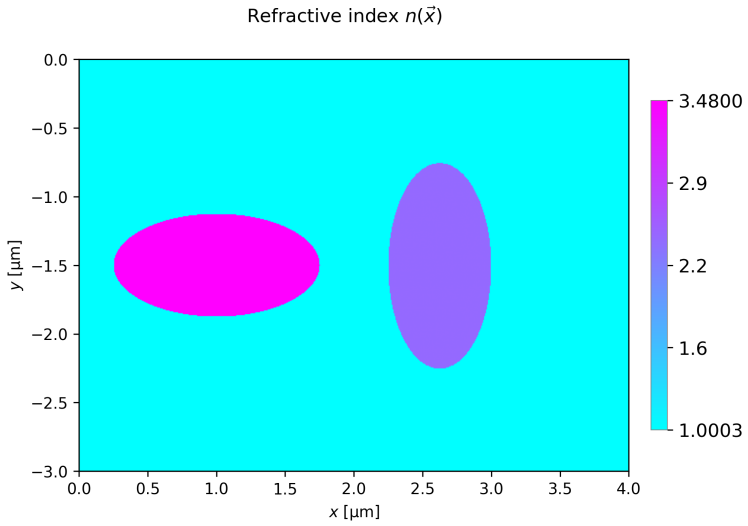
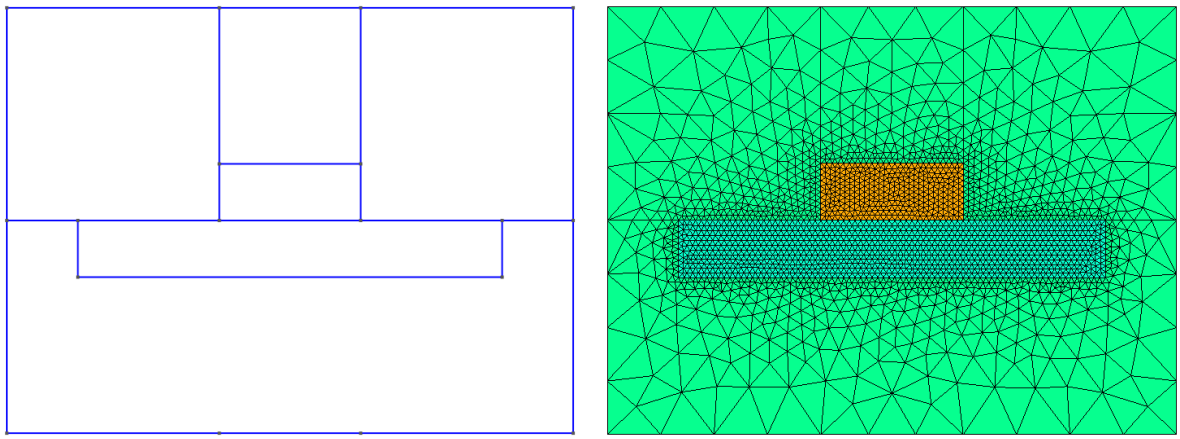



Fig. 5: Coupled circular waveguides using shape `circular` (template `twoincl_msh`). There appears to be a bug here!

### 6.2.3 Rib waveguides

These structures consist of a rib on one or more substrate layers with zero to two coating layers.



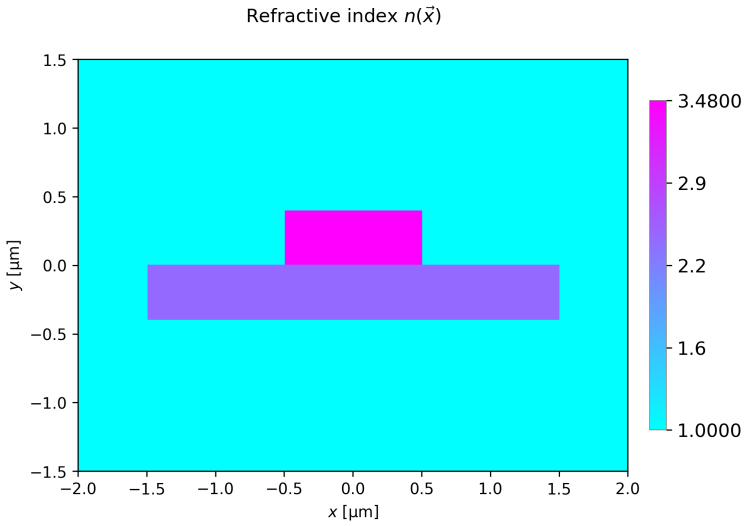



Fig. 6: A conventional rib waveguide using shape `rib` (template `rib`).

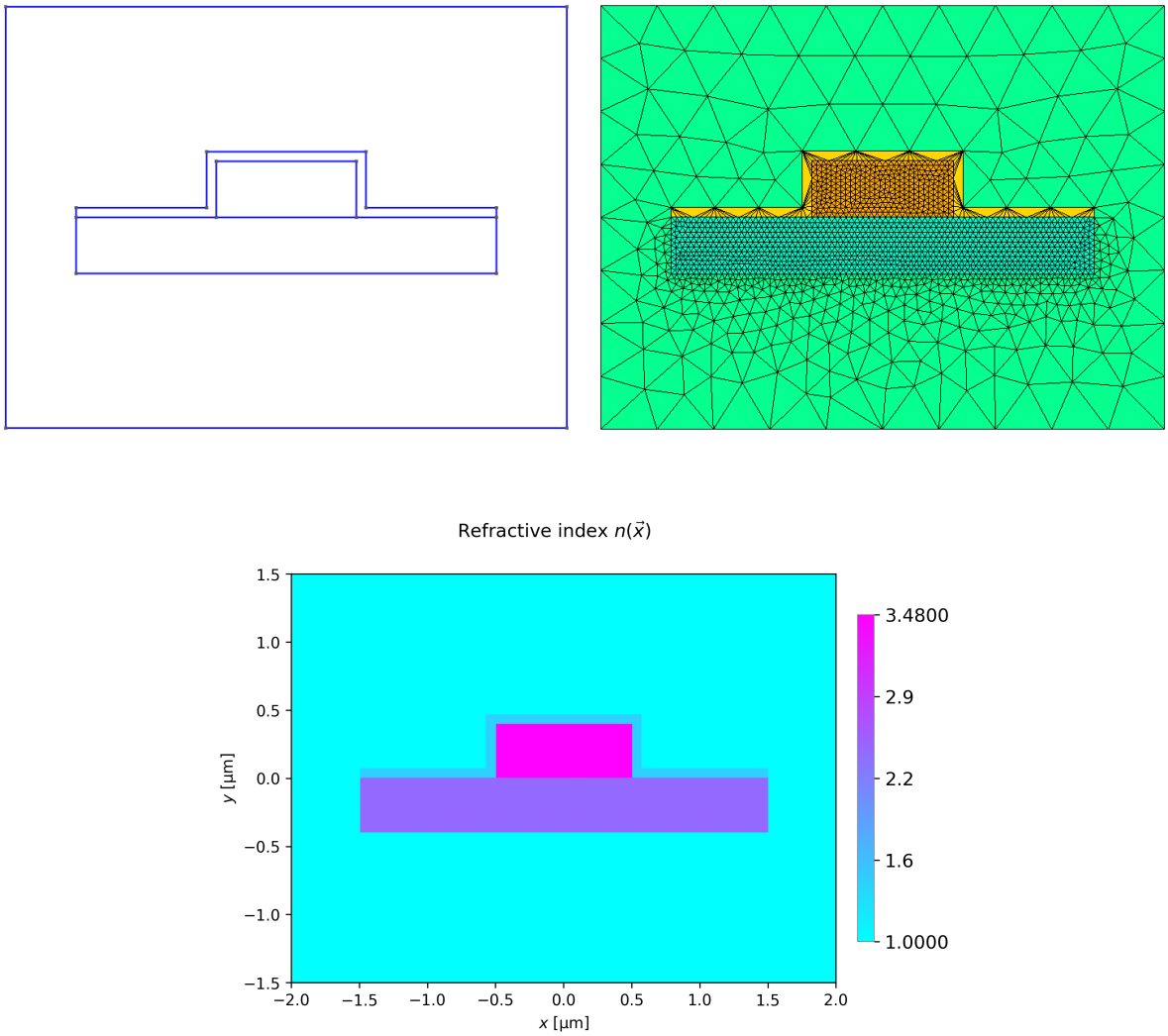



Fig. 7: A coated rib waveguide using shape `rib_coated` (template `rib_coated`).

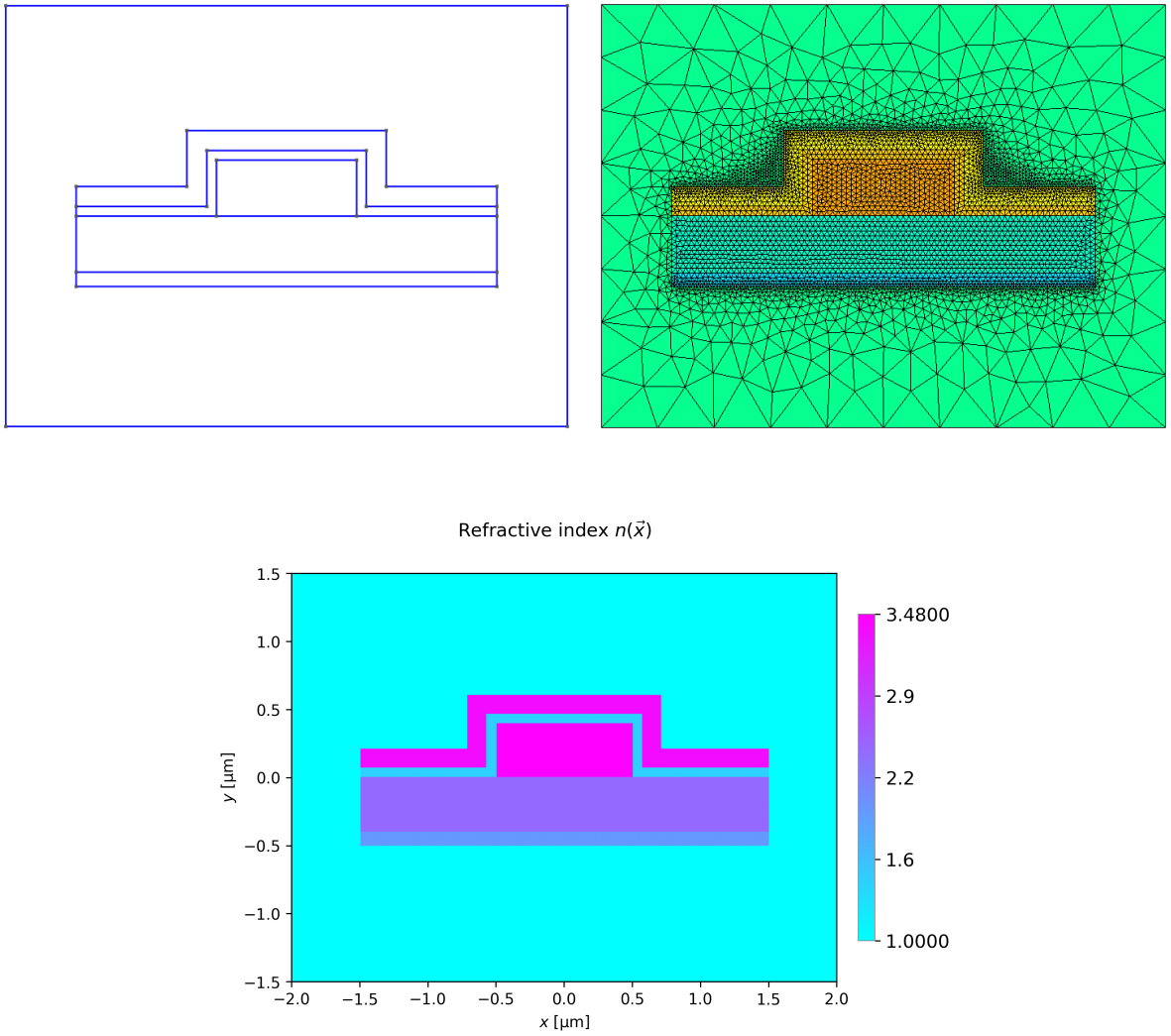
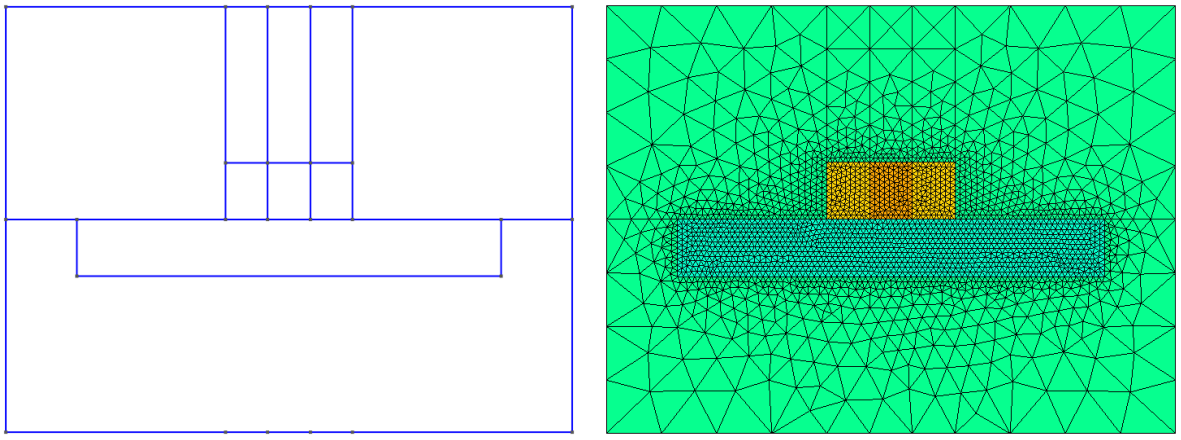



Fig. 8: A rib waveguide on two substrates using shape `rib_double_coated` (template `rib_double_coated`).



### 6.2.4 Engineered rib waveguides

These are examples of more complex rib geometries. These are good examples to study in order to make new designs using the user-specified waveguide and mesh mechanism.

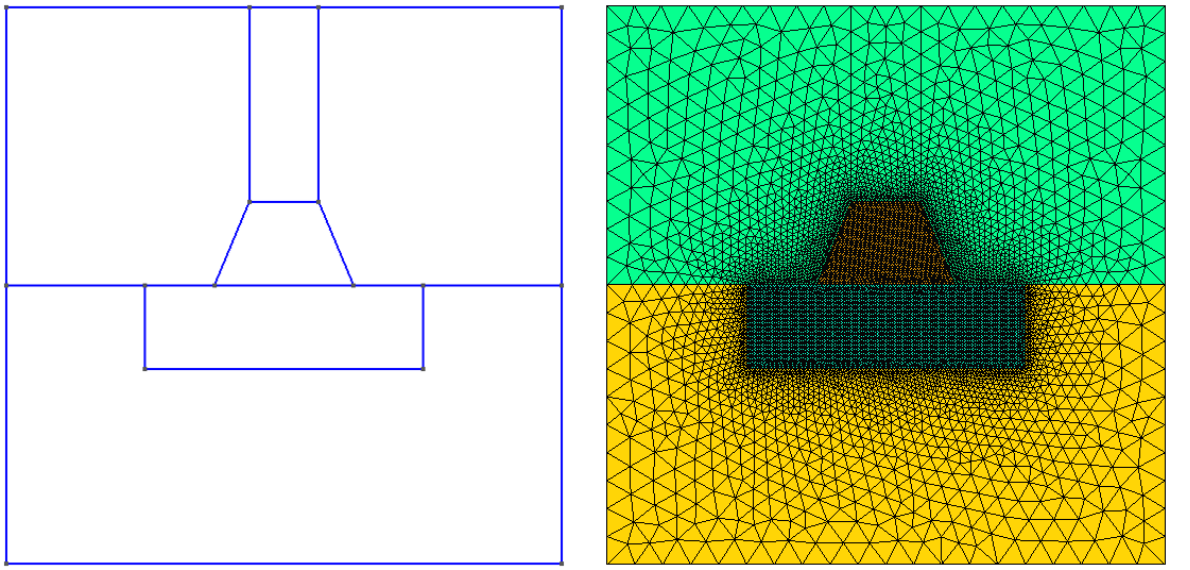


Fig. 9: A trapezoidal rib structure using shape `trapezoidal_rib`.

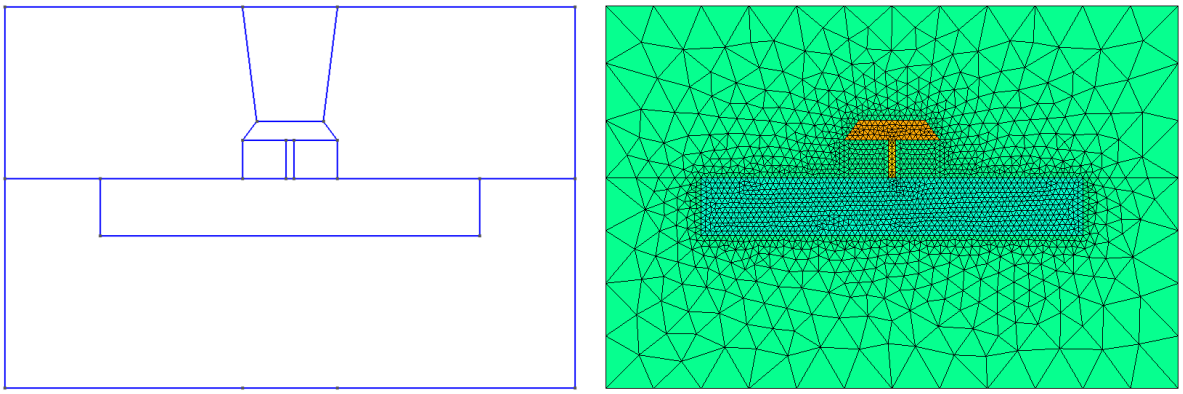


Fig. 10: A supported pedestal structure using shape `pedestal`.

### 6.2.5 Slot waveguides

These slot waveguides can be used to enhance the horizontal component of the electric field in the low index region by the 'slot' effect.

A slot waveguide using shape `slot` (`material_a` is low index) (template `slot`).

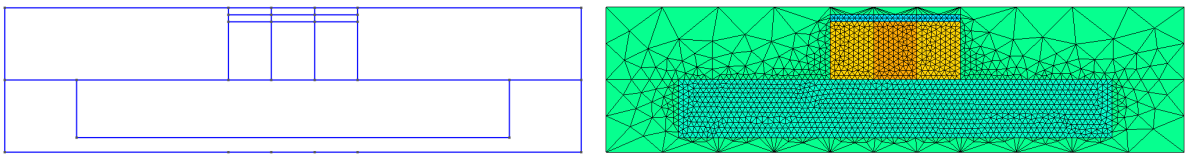


Fig. 11: A coated slot waveguide using shape `slot_coated` (`material_a` is low index) (template `slot_coated`).

### 6.2.6 Layered circular waveguides

These waveguides consist of a set of concentric circular rings of a desired number of layers in either a square or circular outer domain. Note that `inc_a_x` specifies the innermost *diameter*. The subsequent parameters `inc_b_x`, `inc_c_x`, etc specify the annular thickness of each successive layer.

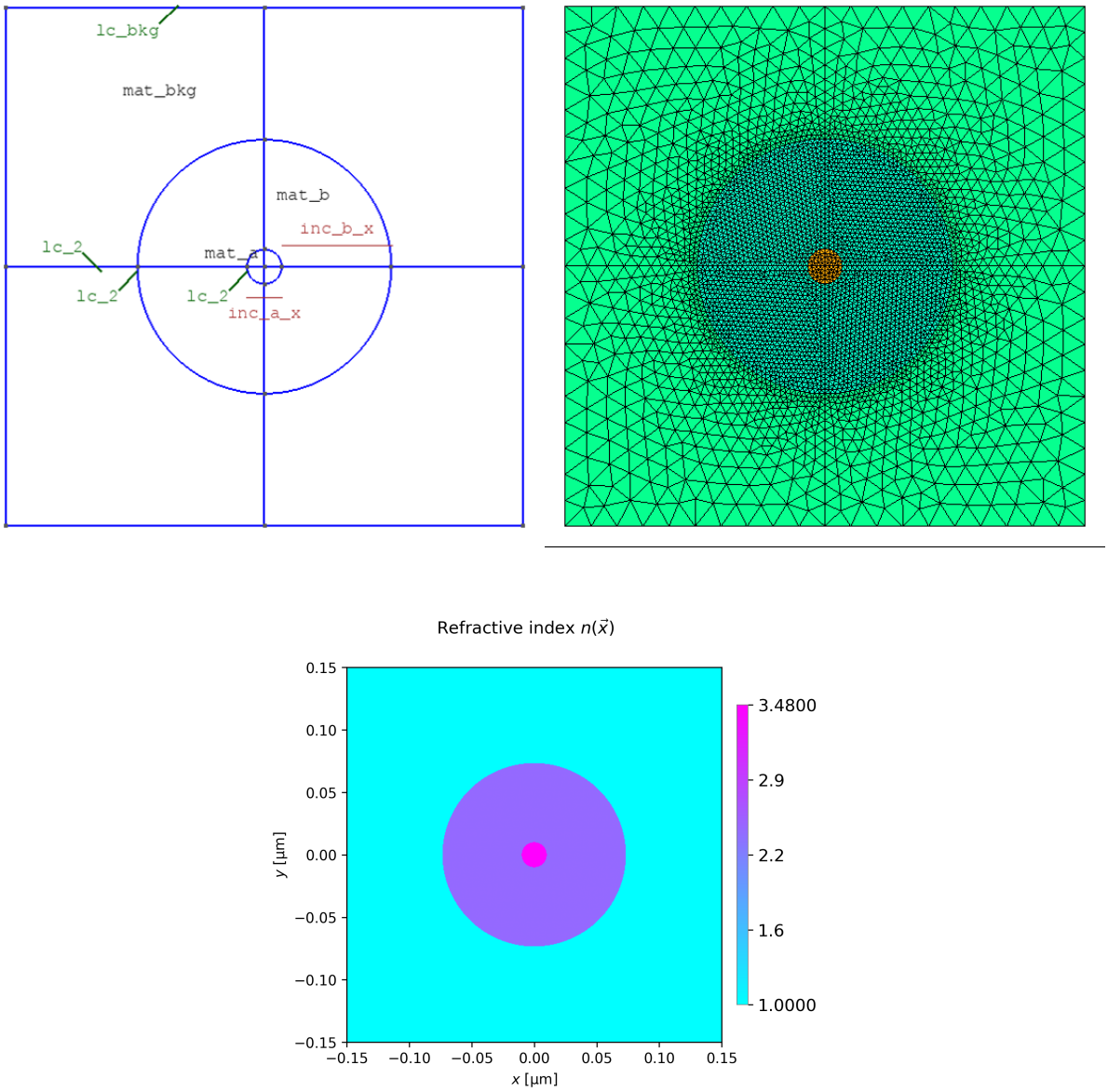



Fig. 12: A two-layered concentric structure with background using shape `onion2` (template `onion2`).

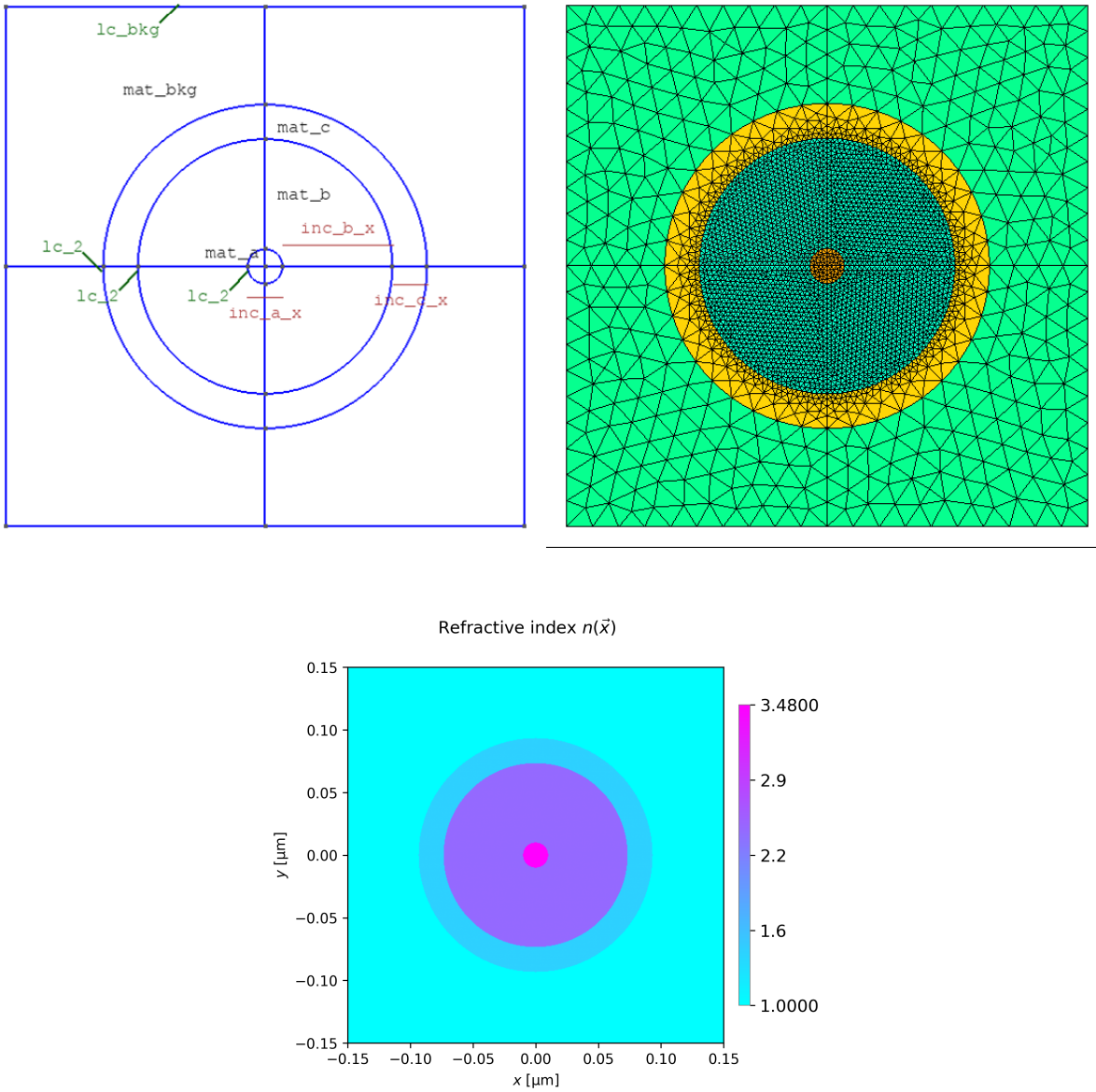



Fig. 13: A three-layered concentric structure with background using shape `onion3` (template `onion3`).

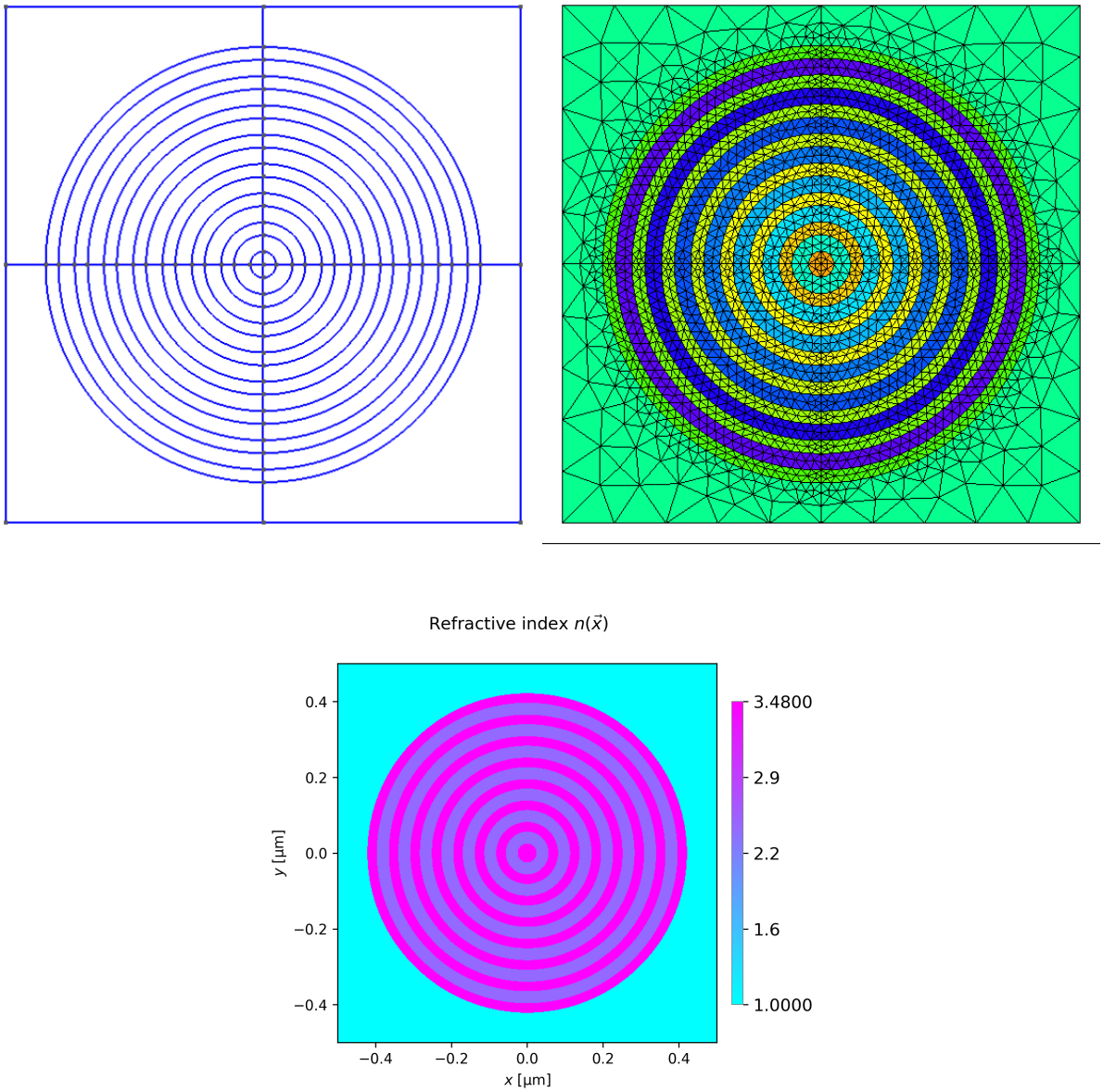



Fig. 14: A many-layered concentric structure using shape `onion` (template `onion`).

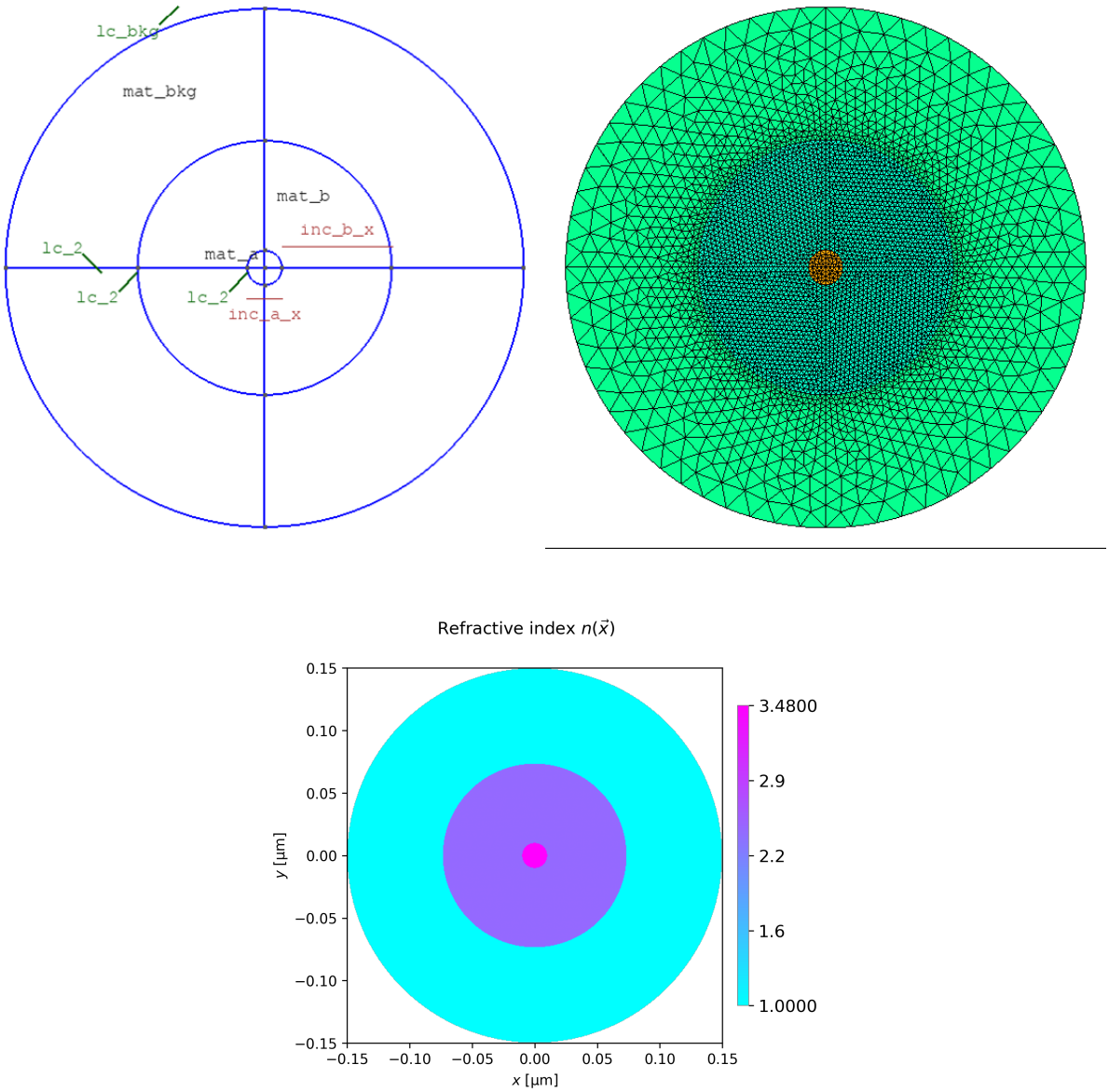



Fig. 15: A two-layered concentric structure with a circular outer boundary using shape `circ_onion2` (template `circ_onion2`).

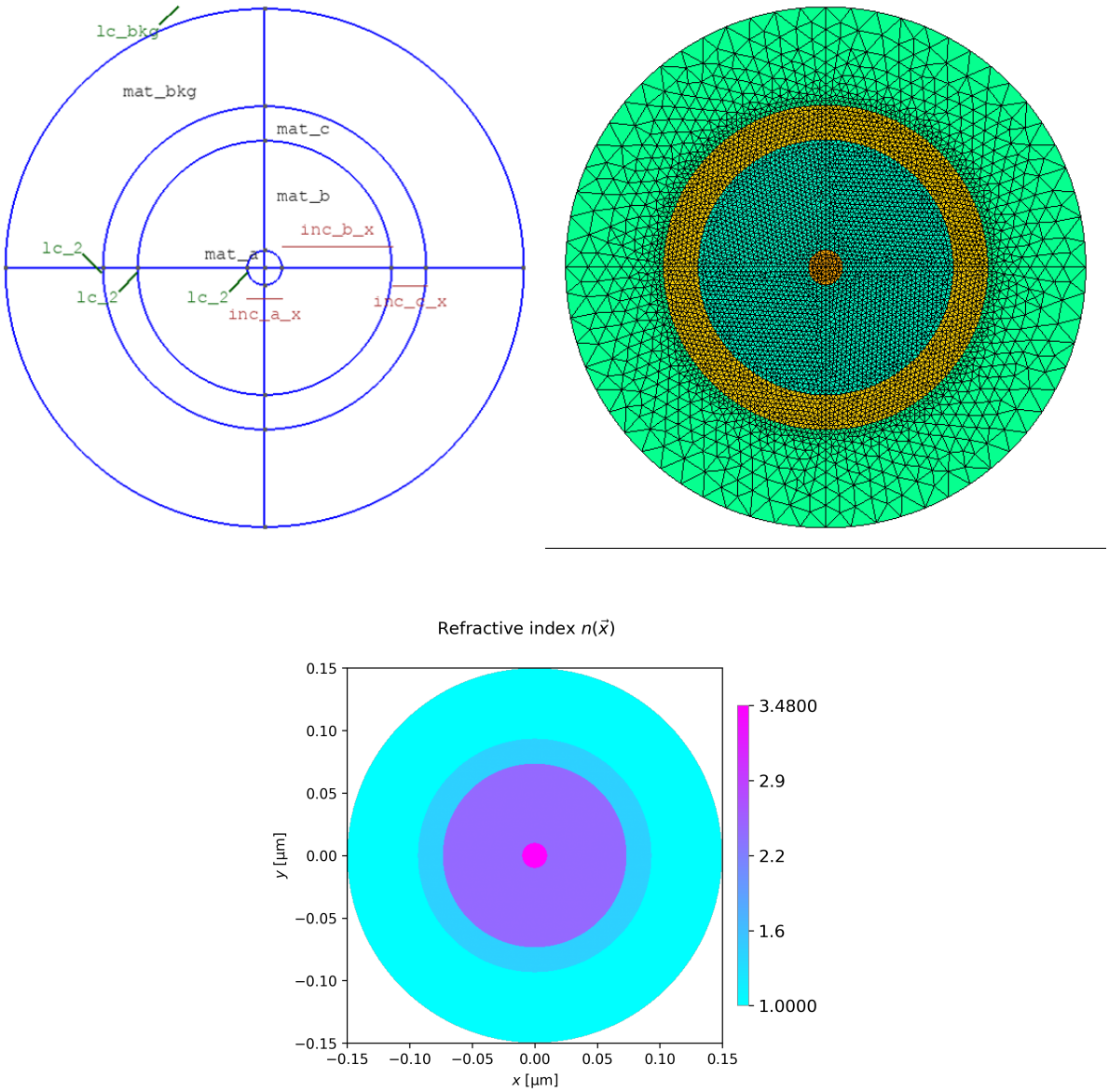



Fig. 16: A three-layered concentric structure with a circular outer boundary using shape `circ_onion3` (template `circ_onion3`).

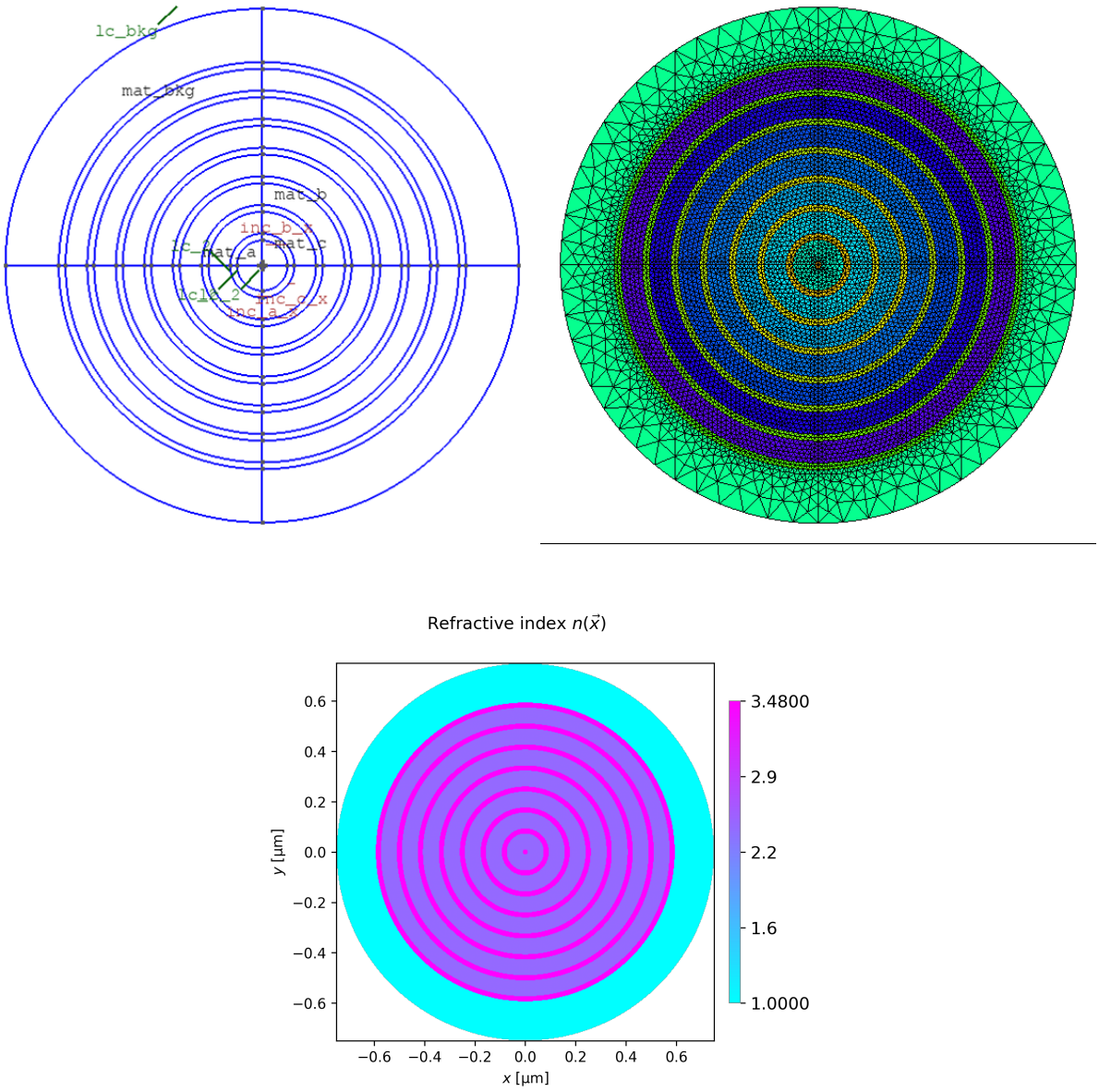



Fig. 17: A many-layered concentric structure with a circular outer boundary using shape `circ_onion` (template `circ_onion`).

## 6.3 User-defined waveguide geometries

Users may incorporate their own waveguide designs fully into NumBAT with the following steps. The `triangular` built-in structure is a helpful model to follow.

- Create a new gmsh template `.geo` file to be placed in `<NumBAT>/backend/msh` that specifies the general structure. Start by looking at the structure of `triangular_msh_template.geo` and some other files to get an idea of the general structure. We'll suppose the file is called `mywaveguide_msh_template.geo` and the template name is thus `mywaveguide`.

  When designing your template, please ensure the following:

  - That you use appropriate-sized parameters for all significant dimensions. This makes it easier to determine if the template structure has the right general shape, even though the precise dimensions will usually be changed through NumBAT calls.
  - That all `Line` elements are unique. In other words do not create two `Line` structure joining the same two points. This will produce designs that look correct, but lead to poorly formed meshes that will fail when NumBAT runs.
  - That all `Line Loop` elements defining a particular region are defined with the same handedness. The natural choice is to go around the loop anti-clockwise. Remember to include a minus sign for any line element that is traversed in the backwards sense.
  - That all regions that define a single physical structure with a common material are grouped together as a single `Surface` and then `Physical Surface`.
  - That the outer boundary is grouped as a `Line Loop` and then a `Physical Line`.
  - That the origin of coordinates is placed in a sensible position, such as a symmetry point close to where you expect the fundamental mode fields to be concentrated. This doesn't actually affect NumBAT calculations but will produce more natural axis scales in output plots.

  You can see all examples of these principles followed in the mesh structures supplied with NumBAT.

- If this is your first, user-defined geometry, copy the file ''user_waveguides.json_template`` in `<NumBAT>/backend/msh/` to `user_waveguides.json` in the same directory. This will ensure that subsequent `git pull` commands will not overwrite your work.
- Open the file `user_waveguides.json` and add a new dictionary element for your new waveguide, copying the general format of the pre-defined example entries.
- Fill in values for the `wg_impl` (the name of the python *file* implementing your waveguide geometry), `wg_class` (the name of the python *class* corresponding to your waveguide) and `inc_shape` (the waveguide *template name*) fields.
  - The value of `inc_shape` will normally be the your chosen template name, in this case `mywaveguide`. The other parameters can be chosen as you wish. It is natural to choose a class name which matches your template name, so perhaps `MyWaveguide`. However, depending on the number of geometries you create, it may be convenient to store all your classes in one python file so the filename for `wg_impl` may be the same for all your entries.
  - The `active` field allows a waveguide to be disabled if it is not yet fully working and you wish to use other NumBAT models in the meantime. You must set `active` to `True` of 1 in order to test your waveguide model.
  - Then save and close this file.
- Open or create the python file you just specified in the `wg_impl` field. This file must be placed in the `<NumBAT>/backend/msh` directory.
  - The python file must include the import line `from usermesh import UserGeometryBase`.
  - Create your waveguide class `MyWaveguide` by subclassing the `UserGeometryBase` class and adding `init_geometry` and `apply_parameters` methods using the `Triangular` class in `builtin_meshes.py` as a model. Both methods must take only `self` as arguments.

- The `init_geometry` method specifies a few values including the name of the template `.geo` file, the number of distinct waveguide components and a short description.
- The `apply_parameters` method is the mechanism for associating standard NumBAT symbols like `inc_a_x`, `slab_a_y`, etc with actual dimensions in your `.geo` file. This is done by string substitution of unique expressions in your `.geo` file using float values evaluated from the NumBAT parameters. Again, look at the examples in the `Triangular` class to see how this works.
- Optionally, you may also add a `draw_mpl_frame` method. This provides a mechanism to draw waveguide outlines onto mode profile images and will be called automatically any time an electromagnetic or elastic mode profile is generated. The built-in waveguides `Circular`, `Rectangular` and `TwoIncl` provide good models for this method.

Designing and implementing a few waveguide structure should not be a daunting task but some steps can be confusing the first time round. If you hit any hiccups or have suggestions for trouble-shooting, please let us know.

## 6.4 Mesh parameters

The parameters `lc_bkg`, `lc_refine_1`, `lc_refine_2` labelled in the above figures control the fineness of the FEM mesh and are set when constructing the waveguide, as discussed in the next chapter. The first parameter `lc_bkg` sets the reference background mesh size, typically as a fraction of the length of the outer boundary edge. A larger `lc_bkg` yields a coarser mesh. Reasonable starting values are `lc_bkg=0.1` (10 mesh points on the outer boundary) to `lc_bkg=0.05` (20 mesh points on the outer boundary).

As well as setting the overall mesh scale with `lc_bkg`, one can also refine the mesh near interfaces and near select points in the domain, as may be observed in the figures in the previous section. This helps to increase the mesh resolution in regions where there the electromagnetic and acoustic fields are likely to be strong and/or rapidly varying. This is achieved using the `lc_refine_n` parameters as follows. At the interface between materials, the mesh is refined to have characteristic length `lc_bkg/lc_refine_1`, therefore a *larger* `lc_refine_1` gives a *finer* mesh by a factor of `lc_refine_1` at these interfaces. The meshing program `Gmsh` automatically adjusts the mesh size to smoothly transition from a point that has one mesh parameter to points that have other meshing parameters. The mesh is typically also refined in the vicinity of important regions, such as in the center of a waveguide, which is done with `lc_refine_2`, which analogously to `lc_refine_1`, refines the mesh size at these points as `lc_bkg/lc_refine_2`.

For more complicated structures, there are additional `lc_refine_<n>` parameters. To see their exact function, look for these expressions in the particular .geo file.

Choosing appropriate values of `lc_bkg`, `lc_refine_1`, `lc_refine_2` is crucial for NumBAT to give accurate results. The appropriate values depend strongly on the type of structure being studied, and so we strongly recommended carrying out a convergence test before delving into new structures (see Tutorial 5 for an example) starting from similar parameters as used in the tutorial simulations.

As will as giving low accuracy, a structure with too coarse a mesh is often the cause of the eigensolver failing to converge in which case NumBAT will terminate with an error. If you encounter such an error, try the calculation again with a slightly smaller value for `lc_bkg`, or slightly higher values for the `lc_refine_n` parameters.

On the other hand, it is wise to begin with relatively coarse meshes. It will be apparent that the number of elements scales roughly *quadratically* with the `lc_refine` parameters and so the run-time increases rapidly as the mesh becomes finer. For each problem, some initial experimentation to identify a mesh resolution that gives reasonable convergence in acceptable simulation is usually worthwhile.

## 6.5 Viewing the mesh

When NumBAT constructs a waveguide, the template `geo` file is converted to a concrete instantiation with the `lc_refine` and geometric parameters adjusted to the requested values. This file is then converted into a `gmsh .msh` file. When exploring new structures and their convergence behaviour, it is a very good idea to view the generated mesh frequently.

You can examine the resolution of your mesh by calling the `plot_mesh(<prefix>)` or `check_mesh()` methods on a waveguide `Structure` object. The first of these functions saves a pair of images of the mesh to a

<prefix>-mesh-annotated.png file in the local directory which can be viewed with your preferred image viewer; the second opens the mesh in a gmsh window (see Tutorial 1 above).

In addition, the .msh file generated by NumBAT in any calculation is stored in <NumBAT>/backend/fortran/msh/build and can be viewed by running the command

```
gmsh <msh_filename>.msh
```

In some error situations, NumBAT will explicitly suggest viewing the mesh and will print out the required command to do so.

CHAPTER

SEVEN

# PROPERTIES OF BULK CRYSTAL ELASTIC MODES

In this chapter we pause to review some of the properties of bulk elastic modes in isotropic and anisotropic crystals and learn how code within NumBAT can explore these properties. While it doesn't directly involve calculation of Brillouin gains, this material is helpful in understanding some of the later examples/tutorials.

This material is most naturally explored interactively and so examples in this chapter are written as Jupyter notebooks stored in the `examples/bulkprops` directory. (See Tutorial 9 for an introduction to NumBAT in Jupyter).

## 7.1 Tutorial 9a - Bulk elastic anisotropy

Although much of the SBS literature assumes that the elastic materials are isotropic, anisotropy of the elastic response can be an important effect. In general, anisotropy is often more significant in elastic physics than electromagnetic physics, because of the more involved tensor nature of the elastic theory. For instance, *cubic* materials such as silicon have an isotropic linear electromagnetic response but an anisotropic elastic linear response.

NumBAT supports arbitrary elastic nonlinearity in calculating elastic modes and the SBS gain of a waveguide. However, even the bulk elastic wave properties of anistropic materials is quite complex. This tutorial explores some of these effects.

This exercise is most naturally performed interactively and so this example is written as a Jupyter notebook (see Tutorial 9 for an introduction to NumBAT in Jupyter).

### 7.1.1 Theory

Bulk wave modes in linear elastic materials are found as eigen-solutions of the elastic wave equation for a uniform material.
We start from the elastic wave equation

$$\nabla \cdot \bar{T} + \omega^2 \rho(x,y) \vec{U} = 0,$$

where $\bar{T}$ is the stress tensor and $\vec{U}$ is the displacement field. (In this notebook, we use overlined quantities to denote a general tensor, and drop the overline when using index notation, so $\bar{T} \leftrightarrow T_{ij}$.)

Using the constitutive equation

$$\bar{T} = \bar{c} : \bar{S} \qquad \leftrightarrow \qquad T_{ij} = c_{ijkl} S_{kl},$$

where $\bar{c}$ is the stiffness tensor and $S_{kl}=:nbsphinx-math:*frac{1}{2}*(:nbsphinx-math:frac{partial U_k}{partial r_l}+:nbsphinx-math:*frac{partial U_l}{partial r_k}*)$ the strain tensor, we find

$$\nabla \cdot (\bar{c} : \bar{S}) + \omega^2 \rho(x,y) \vec{U} = 0$$
$$\nabla \cdot (\bar{c} : \nabla_s \vec{U}) + \omega^2 \rho(x,y) \vec{U} = 0,$$

where $\nabla_S$ denotes the *symmetric gradient*, so that $S_{ij} = (\nabla_S \vec{U})_{ij}$.

### 7.1.2 Bulk wave modes

Looking for plane wave solutions of the form

$$\vec{U} = \vec{u}e^{i(\vec{q}\cdot\vec{r}-\Omega t)} + \vec{u}^* e^{-i(\vec{q}\cdot\vec{r}-\Omega t)},$$

leads to the 3x3 matrix eigenvalue equation (see Auld. vol 1, chapter 7)

$$q^2\Gamma\vec{u} = \rho\Omega^2\vec{u},$$

or in index form

$$(q^2\Gamma_{ij} - \rho\Omega^2\delta_{ij})u_j = 0,$$

which is known as the *Christoffel* equation. Here the wavevector $\vec{q}$ has been written $\vec{q} = q\hat{\kappa}$ in terms of the unit vector $\hat{\kappa}$.

The 3x3 matrix operator $\Gamma$ is most conveniently written using the compact Voigt notation as follows. We define the matrix

$$\mathrm{M} = \begin{bmatrix} \kappa_x & 0 & 0 & 0 & \kappa_z & \kappa_y \\ 0 & \kappa_y & 0 & \kappa_z & 0 & \kappa_x \\ 0 & 0 & \kappa_z & \kappa_y & \kappa_x & 0 \end{bmatrix}.$$

Then one can check by direct multiplication that $\Gamma$ has the form

$$\Gamma(\vec{\kappa}) = \mathrm{M}C_{IJ}\mathrm{M}^t,$$

where $C_{IJ}$ is the 6x6 Voigt matrix for the stiffness tensor (see Auld chapter 2).

Since the stiffness is invariably treated as frequency independent, we can rewrite the Christoffel equation as

$$\left(\frac{1}{\rho}\Gamma_{ij} - \frac{\Omega^2}{q^2}\delta_{ij}\right)u_j = 0,$$

and identify the eigenvalue as the square of the phase speed $v = \Omega/q$:

$$\left(\frac{1}{\rho}\Gamma_{ij}(\vec{\kappa}) - v^2\delta_{ij}\right)u_j = 0.$$

If we neglect the viscosity, $\Gamma$ is a real symmetric matrix, so we are guaranteed to find three propagating wave modes with real phase velocities $v_i$ and orthogonal polarisation vectors $\vec{u}_i$.

In isotropic materials, the Christoffel equation has the expected solutions of one longitudinal wave, and two slower shear waves. In anisotropic materials, the polarisations can be more complicated. However, as $\Gamma$ is a symmetric matrix, the three wave modes are always orthogonal.

### 7.1.3 Group velocity

Continuing to neglect any linear wave damping, we can identify the *group velocity*

$$\vec{v}_g \equiv \nabla_{\vec{q}}\Omega,$$

while the *energy velocity* $\vec{v}_e$, defined as the ratio of the power flux and the energy density, is

$$\vec{v}_g \equiv \frac{P_e}{u_e} = \frac{-\frac{1}{2}\vec{v}\cdot\bar{T}}{\bar{\bar{S}}:\bar{\bar{C}}:\bar{\bar{S}}}.$$

In this way, we can find both the phase velocity and group velocity as functions of the wavevector direction $\vec{\kappa}$. In the excellent approximation of zero material dispersion, these two velocities are independent of the wave frequency $\Omega$. This is *not* true in waveguides, where the spatial confinement does lead to signficant dispersion.

### 7.1.4 Wave surfaces

To understand the directional dependence of the different wave properties, it is common to plot several scalar quantities

- the *slowness surface*, which is the reciprocal of the wave speed $\frac{1}{v_p(\vec{\kappa})}$
- the *normal* or *phase velocity* surface, which is simply the wave speed function $v_p(\vec{\kappa})$
- the *ray surface*, which is the magnitude of the group velocity $|\vec{v}_g(\vec{\kappa})|$

Note that while both the phase and group velocities are vectors, since the phase velocity is everywhere parallel to the wavevector direction $\vec{\kappa}$, it is convenient to simply refer to the wave speed $v_p$ written as a scalar. We can't do this with the group velocity, which for anisotropic materials, is not generally parallel to the wavevector.

[1]:
```
%load_ext autoreload
%autoreload 3

import sys
import numpy as np
from IPython.display import Image, display

sys.path.append("../backend")
#import numbattools
import materials
```

### 7.1.5 Wave properties of isotropic materials

Let's start by calculating the above properties for an isotropic medium, say fused silica. We create the material and print out a few of its basic properties.

[2]:
```
mat_a = materials.make_material("SiO2_2021_Poulton")

print(mat_a, '\n')

print(mat_a.elastic_properties())
```

```
Material: SiO2
  File: SiO2_2021_Poulton
  Source: Poulton
  Date: 2021

Elastic properties of material SiO2_2021_Poulton
  Density:        2200.000 kg/m^3
  Ref. index:     1.4500+0.0000j
  Crystal class:  Isotropic
  c11:            78.500 GPa
  c12:            16.100 GPa
  c44:            31.200 GPa
  Young's mod E:  73.020 GPa
  Poisson ratio:  0.170
  Velocity long.: 5973.426 m/s
  Velocity shear: 3765.875 m/s
  Velocity Rayleigh: 3411.154 m/s
```

Observe that this material has a *crystal class* of *Isotropic*, and that its stiffness values satisfy the constraint $c_{44} = (c_{11} - c_{12})/2$ which holds for any isotropic material.

Further, being isotropic, it has a well-defined Young's modulus and Poisson ratio. In fact, for isotropic materials, NumBAT allows the material properties to be specified in terms of those quantities rather than the stiffness values if desired.

The longitudinal and shear phase speeds are given for propagation along $z$ with $\vec{\kappa} = (0, 0, 1)$. Of course for this isotropic material, the phase speeds are actually the same in every direction.

We can examine the complete material tensors directly and confirm that they have the expected forms for an isotropic material:

[4]:
```
print('\n\nStiffness:', mat_a.stiffness_c_IJ)

print('\n\nPhotoelasticity:', mat_a.photoel_p_IJ)
```

```
Stiffness:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[78.5 16.1 16.1  0.   0.   0. ]
     [16.1 78.5 16.1  0.   0.   0. ]
     [16.1 16.1 78.5  0.   0.   0. ]
     [ 0.   0.   0.  31.2  0.   0. ]
     [ 0.   0.   0.   0.  31.2  0. ]
     [ 0.   0.   0.   0.   0.  31.2]]


Photoelasticity:
 Photoelasticity p_IJ., unit: dimensionless.
   Voigt 4-tensor:
    [[ 0.121  0.271  0.271  0.     0.     0.   ]
     [ 0.271  0.121  0.271  0.     0.     0.   ]
     [ 0.271  0.271  0.121  0.     0.     0.   ]
     [ 0.     0.     0.    -0.075  0.     0.   ]
     [ 0.     0.     0.     0.    -0.075  0.   ]
     [ 0.     0.     0.     0.     0.    -0.075]]
```

### 7.1.6 Crystal rotations

For aniostropic crystals, different directions are not equivalent.

NumBAT supports several mechanisms for applying crystal rotations to materials. This allows modelling of waveguides fabricated using different *cuts* of the same material.

For an isotropic material, a crystal rotation should have no consequential effect. Let's check that this holds.

The following code creates a copy of the original material, and then rotates its crystal properties by an angle $\pi/3$ around the direction of the vector $\vec{n} = [1.0, 1.0, 1.0]$ (which need not be normalised) in the positive right-hand sense.

[ ]:
```
mat_b = mat_a.copy()

nvec = np.array([1.0,1.0,1.0])
phi = np.pi/3.

mat_b.rotate(nvec, phi)

print(mat_b.elastic_properties())

print(mat_b.stiffness_c_IJ)

# Measure the difference in the original and rotated stiffness tensors
```

(continues on next page)

(continued from previous page)

```
err = np.linalg.norm(mat_b.stiffness_c_IJ.mat - mat_a.stiffness_c_IJ.mat)/np.abs(mat_
↪a.stiffness_c_IJ.mat).max()
print(f'\n\n Relative change in stiffness tensor: {err:.4e}')
```

```
Elastic properties of material SiO2_2021_Poulton
  Density:        2200.000 kg/m^3
  Ref. index:     1.4500+0.0000j
  Crystal class:  Isotropic
  c11:            78.500 GPa
  c12:            16.100 GPa
  c44:            31.200 GPa
  Young's mod E:  73.020 GPa
  Poisson ratio:  0.170
  Velocity long.: 5973.426 m/s
  Velocity shear: 3765.875 m/s

 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[7.8500e+01 1.6100e+01 1.6100e+01 9.5367e-16 5.7220e-15 4.7684e-15]
     [1.6100e+01 7.8500e+01 1.6100e+01 5.9605e-15 1.9073e-15 1.2398e-14]
     [1.6100e+01 1.6100e+01 7.8500e+01 4.7684e-15 7.6294e-15 9.5367e-16]
     [2.8610e-15 8.8215e-15 4.7684e-15 3.1200e+01 1.9073e-15 2.8610e-15]
     [1.1444e-14 2.3842e-15 7.6294e-15 2.8610e-15 3.1200e+01 9.5367e-16]
     [2.8610e-15 6.9141e-15 9.5367e-16 3.3379e-15 1.9073e-15 3.1200e+01]]


 Relative change in stiffness tensor: 6.2323e-16
```

We can see from the last line that all the properties are unchanged to numerical precision.

### Crystal orientation diagram

However, not *everything* is identical in NumBAT's representations of the original and rotated material, even though the two materials are physically the same .

NumBAT materials include internal *crystal axes* $\{\hat{c}_x, \hat{c}_y, \hat{c}_z\}$ that are a distinct concept from the *waveguide* (laboratory) axes $\{\hat{x}, \hat{y}, \hat{z}\}$. In NumBAT calculations, the waveguide cross-section always lies in the $\hat{x} - \hat{y}$ laboratory coordinate plane and the propagation direction is always along $\hat{z}$. To ensure a right-handed coordinate set, $\hat{z}$ should be viewed as pointing *out* of the screen. (It's not often that we need to worry about the distinction between propagation in or out of the screen, but it does play a role in determining the correct relative signs of the different field components).

The crystal axes $\{\hat{c}_x, \hat{c}_y, \hat{c}_z\}$ define the intrinsic directions for specifying the material stiffness, photoelastic and viscosity tensors. When a material is first loaded from its `json` file, the two sets of axes coincide, so that the Voigt indices $1..6$ correspond to the pairs $xx$, $yy$, $zz$, $xz$, $yz$, $xy$. When a rotation is performed, it is the *crystal* axes that change, so that the anisotropic material properties are "dragged through" the stationary waveguide structure. This can be a little confusing but is simpler than allowing the waveguide structure and propagation direction to rotate instead.

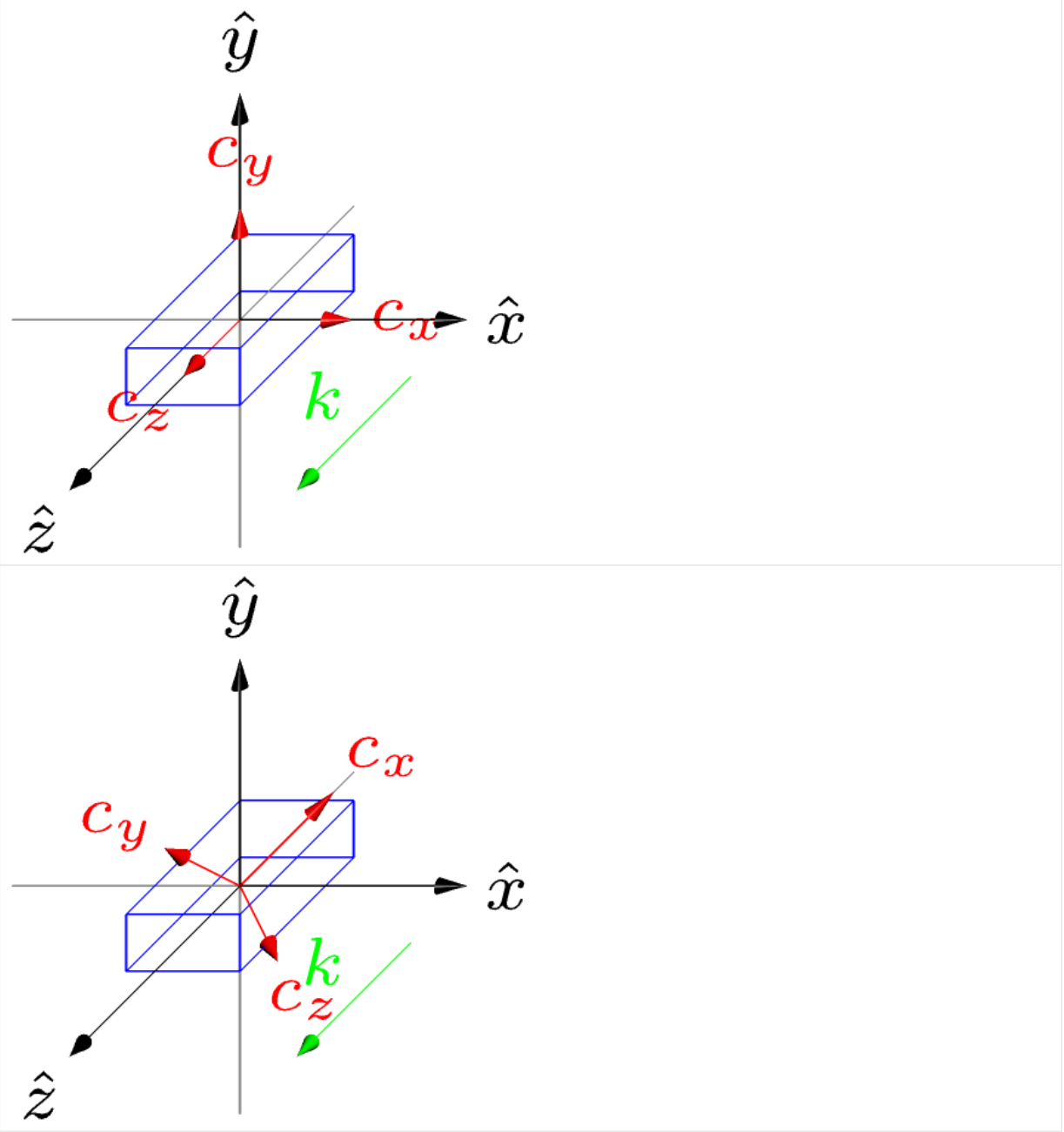
To help ensure the correct orientation is selected, both sets of axes can be plotted together using the `Material.make_crystal_axes_plot` as follows:

[6]:
```
prefa = 'tmp_mata'
prefb = 'tmp_matb'

mat_a.make_crystal_axes_plot(prefa)
mat_b.make_crystal_axes_plot(prefb)

display(Image(prefa+'-crystal.png', width=300))
```

(continues on next page)

(continued from previous page)

```
display(Image(prefb+'-crystal.png', width=300))
```




Observe that the crystal axes for the first material are in the default orientation aligned with the laboratory axes. The crystal axes for the second material have been rotated as described above. The blue box gives a sense of the orientation of the waveguide with propagation out of the screen along $\vec{k} \propto \hat{z}$.

### 7.1.7 Anistropic materials

We now turn to anisotropic materials where the specific crystal orientation begins to matter.

For we consider GaAs, which is a *cubic* material with symmetry group $4'3m$.

[4]:
```
mat_gaas = materials.make_material("GaAs_1970_Auld")
```

(continues on next page)

(continued from previous page)

```
print(mat_gaas, '\n')

print(mat_gaas.elastic_properties())
```

```
Material: GaAs [100]
  File: GaAs_1970_Auld
  Source: Auld
  Date: 1970

Elastic properties of material GaAs_1970_Auld
  Density:        5307.000 kg/m^3
  Ref. index:     3.3702+0.0000j
  Crystal class:  Cubic
  Crystal group:  4'3m
  Stiffness c_IJ:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[118.8  53.8  53.8   0.    0.    0. ]
     [ 53.8 118.8  53.8   0.    0.    0. ]
     [ 53.8  53.8 118.8   0.    0.    0. ]
     [  0.    0.    0.   59.4   0.    0. ]
     [  0.    0.    0.    0.   59.4   0. ]
     [  0.    0.    0.    0.    0.   59.4]]

  Wave mode 1: v_p=4.7313 km/s,  |v_g|=4.7313 km/s,  u_j=[ 0.0000  0.0000  1.0000], ␣
↪v_g=[ 0.0000  0.0000  4.7313] km/s
  Wave mode 2: v_p=3.3456 km/s,  |v_g|=3.3456 km/s,  u_j=[ 1.0000  0.0000  0.0000], ␣
↪v_g=[ 0.0000  0.0000  3.3456] km/s
  Wave mode 3: v_p=3.3456 km/s,  |v_g|=3.3456 km/s,  u_j=[ 0.0000  1.0000  0.0000], ␣
↪v_g=[ 0.0000  0.0000  3.3456] km/sPiezoelectric properties:
 Piezo effects enabled

 Strain piezo coefficient d_iJ, unit: pC/N.
   Voigt 3-tensor:
    [[0.  0.  0.  2.6 0.  0. ]
     [0.  0.  0.  0.  2.6 0. ]
     [0.  0.  0.  0.  0.  2.6]]

 Stress piezo coefficient e_iJ, unit: C/m^2.
   Voigt 3-tensor:
    [[0.     0.     0.     0.1544 0.     0.    ]
     [0.     0.     0.     0.     0.1544 0.    ]
     [0.     0.     0.     0.     0.     0.1544]]

 Dielectric tensor epsS_ij
    [[11.3582+0.j  0.    +0.j  0.    +0.j]
     [ 0.    +0.j 11.3582+0.j  0.    +0.j]
     [ 0.    +0.j  0.    +0.j 11.3582+0.j]]
```

Look at the lines reporting the three wave mode velocities and displacement eigenvectors.

With the default orientation, the separation into longitudinal and shear modes is simple, and for each mode, the phase and group velocities are identical. Moreover, as expected the longitudinal mode is oriented along $z$ and the degenerate shear modes lie in the $x$-$y$ plane.

### Rotating an anistropic material

However, things become more interesting if we start rotating the crystal.

First, let's make a $\pi/2$ rotation around the $y$ axis:

[5]:
```
nvec = np.array([0.0,1.0,0.0])
phi = np.pi/2.

mat_gaas2= mat_gaas.copy()

mat_gaas2.rotate(nvec, phi)

print(mat_gaas2.elastic_properties())
```

```
Elastic properties of material GaAs_1970_Auld
  Density:        5307.000 kg/m^3
  Ref. index:     3.3702+0.0000j
  Crystal class:  Cubic
  Crystal group:  4'3m
  Stiffness c_IJ:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[ 1.1880e+02  5.3800e+01  5.3800e+01  0.0000e+00 -3.2943e-15  0.0000e+00]
     [ 5.3800e+01  1.1880e+02  5.3800e+01  0.0000e+00  0.0000e+00  0.0000e+00]
     [ 5.3800e+01  5.3800e+01  1.1880e+02  0.0000e+00  3.2943e-15  0.0000e+00]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  5.9400e+01  0.0000e+00  0.0000e+00]
     [-3.2943e-15  0.0000e+00  3.2943e-15  0.0000e+00  5.9400e+01  0.0000e+00]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  5.9400e+01]]

  Wave mode 1: v_p=4.7313 km/s,  |v_g|=4.7313 km/s,  u_j=[ 0.0000  0.0000  1.0000], ␣
↪v_g=[ 0.0000  0.0000  4.7313] km/s
  Wave mode 2: v_p=3.3456 km/s,  |v_g|=3.3456 km/s,  u_j=[ 1.0000  0.0000  0.0000], ␣
↪v_g=[-0.0000  0.0000  3.3456] km/s
  Wave mode 3: v_p=3.3456 km/s,  |v_g|=3.3456 km/s,  u_j=[ 0.0000  1.0000  0.0000], ␣
↪v_g=[ 0.0000  0.0000  3.3456] km/sPiezoelectric properties:
 Piezo effects enabled

 Strain piezo coefficient d_iJ, unit: pC/N.
   Voigt 3-tensor:
    [[0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00]
     [0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 1.592e-16]
     [0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00 0.000e+00]]

 Stress piezo coefficient e_iJ, unit: C/m^2.
   Voigt 3-tensor:
    [[0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00]
     [0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 9.4567e-18]
     [0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00]]

 Dielectric tensor epsS_ij
    [[11.3582+0.j  0.    +0.j  0.    +0.j]
     [ 0.    +0.j 11.3582+0.j  0.    +0.j]
     [ 0.    +0.j  0.    +0.j 11.3582+0.j]]
```

Nothing changes! Since the crystal symmetry is cubic, the $\pi/2$ rotation around a crystal axis has left the material unchanged and all the wave properties are the same.

Now let's try a $\pi/4$ rotation around the $y$ axis:

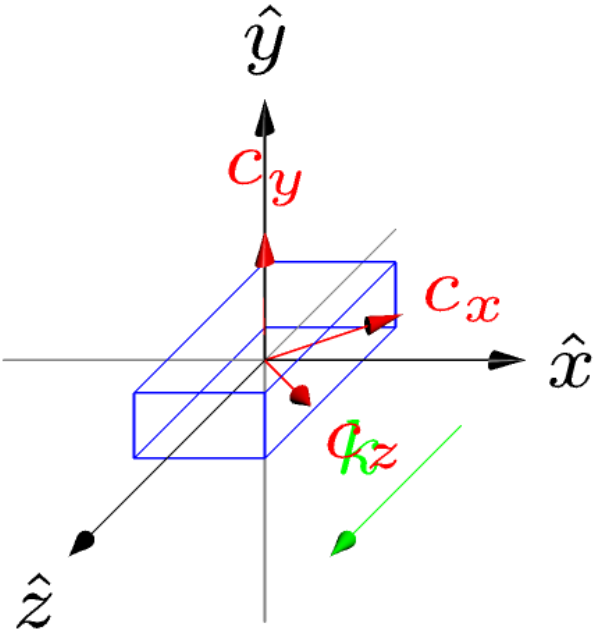
[51]:
```
nvec = np.array([0.0,1.0,0.0])
phi = np.pi/4.

mat_gaas2= mat_gaas.copy()

mat_gaas2.rotate(nvec, phi)

print(mat_gaas2.elastic_properties())
```

```
Elastic properties of material GaAs_1970_Auld
  Density:        5307.000 kg/m^3
  Ref. index:     3.3702+0.0000j
  Crystal class:  Cubic
  Crystal group:  4'3m
  Stiffness c_IJ:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[ 1.4570e+02  5.3800e+01  2.6900e+01  0.0000e+00  7.6294e-15  0.0000e+00]
     [ 5.3800e+01  1.1880e+02  5.3800e+01  0.0000e+00  0.0000e+00  0.0000e+00]
     [ 2.6900e+01  5.3800e+01  1.4570e+02  0.0000e+00 -7.6294e-15  0.0000e+00]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  5.9400e+01  0.0000e+00  0.0000e+00]
     [-3.8147e-15  0.0000e+00 -1.1444e-14  0.0000e+00  3.2500e+01  0.0000e+00]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  5.9400e+01]]

  Wave mode 1: v_p=5.2397 km/s,  |v_g|=5.2397 km/s,  u_j=[-0.0000  0.0000  1.0000], ␣
↪v_g=[-0.0000  0.0000  5.2397] km/s
  Wave mode 2: v_p=3.3456 km/s,  |v_g|=3.3456 km/s,  u_j=[ 0.0000  1.0000  0.0000], ␣
↪v_g=[ 0.0000  0.0000  3.3456] km/s
  Wave mode 3: v_p=2.4747 km/s,  |v_g|=2.4747 km/s,  u_j=[ 1.0000  0.0000  0.0000], ␣
↪v_g=[ 0.0000  0.0000  2.4747] km/s
Piezoelectric properties:
 Piezo effects enabled

 Strain piezo coefficient d_iJ, unit: pC/N.
   Voigt 3-tensor:
    [[0.     0.     0.     0.     0.     0.    ]
     [0.     0.     0.     0.     0.     1.8385]
     [0.     0.     0.     0.     0.     0.    ]]

 Stress piezo coefficient e_iJ, unit: C/m^2.
   Voigt 3-tensor:
    [[0.     0.     0.     0.     0.     0.    ]
     [0.     0.     0.     0.     0.     0.1092]
     [0.     0.     0.     0.     0.     0.    ]]

 Dielectric tensor epsS_ij
    [[11.3582+0.j  0.    +0.j  0.    +0.j]
     [ 0.    +0.j 11.3582+0.j  0.    +0.j]
     [ 0.    +0.j  0.    +0.j 11.3582+0.j]]
```

Observe that the polarisation states indicated by the components of the $\vec{u}$ vectors are unchanged: there is a longitudinal mode oriented along $z$ and two shear modes with vibrations in the $x$-$y$ plane. Moreover, the group velocity of each mode is still aligned with the wavevector $\vec{q} \propto \hat{z}$.

However the two shear modes are no longer degenerate: they have different phase and group speeds, with the $x$-polarised mode having slowed by about 26%. As the crystal axis diagram below shows, this makes sense. The polarisation vectors of the two shear modes point along $\hat{x}$ and $\hat{y}$, but these two laboratory directions are no longer equivalent directions in the crystal.

[8]:
```
prefg='tmp_gaas2'
mat_gaas2.make_crystal_axes_plot(prefg)

display(Image(prefg+'-crystal.png', width=300))
```




Things get really interesting if we apply a rotation that is not commenusare with the crytal symmetries: a positive $\pi/3$ rotation around the $[1,1,1]$ direction:

[ ]:
```
nvec = np.array([1.0,1.0,1.0])
phi = np.pi/3.

mat_gaas3= mat_gaas.copy()
mat_gaas3.rotate(nvec, phi)
print(mat_gaas3.elastic_properties())
```

```
Elastic properties of material GaAs_1970_Auld
  Density:         5307.000 kg/m^3
  Ref. index:      3.3702+0.0000j
  Crystal class:   Cubic
  Crystal group:   4'3m
  Stiffness c_IJ:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[150.6815  37.8593  37.8593   7.9704  -3.9852  -3.9852]
     [ 37.8593 150.6815  37.8593  -3.9852   7.9704  -3.9852]
     [ 37.8593  37.8593 150.6815  -3.9852  -3.9852   7.9704]
     [  7.9704  -3.9852  -3.9852  43.4593   7.9704   7.9704]
     [ -3.9852   7.9704  -3.9852   7.9704  43.4593   7.9704]
     [ -3.9852  -3.9852   7.9704   7.9704   7.9704  43.4593]]

  Wave mode 1: v_p=5.3341 km/s,  |v_g|=5.3484 km/s,  u_j=[-0.0400 -0.0400  0.9984], ␣
↪v_g=[-0.2763 -0.2763  5.3341] km/s
  Wave mode 2: v_p=3.1034 km/s,  |v_g|=3.3219 km/s,  u_j=[ 0.7060  0.7060  0.0565], ␣
↪v_g=[ 0.8378  0.8378  3.1034] km/s
  Wave mode 3: v_p=2.5860 km/s,  |v_g|=2.6583 km/s,  u_j=[-0.7071  0.7071 -0.0000], ␣
↪v_g=[-0.4356 -0.4356  2.5860] km/s
Piezoelectric properties:
```

(continues on next page)

(continued from previous page)

```
Piezo effects enabled

Strain piezo coefficient d_iJ, unit: pC/N.
  Voigt 3-tensor:
   [[0.     0.     0.     0.     0.     0.1926]
    [0.     0.     0.     0.     0.     0.7704]
    [0.     0.     0.     0.     0.     0.1926]]

Stress piezo coefficient e_iJ, unit: C/m^2.
  Voigt 3-tensor:
   [[0.     0.     0.     0.     0.     0.0114]
    [0.     0.     0.     0.     0.     0.0458]
    [0.     0.     0.     0.     0.     0.0114]]

Dielectric tensor epsS_ij
   [[1.1358200e+01+0.j 8.8817842e-16+0.j 8.8817842e-16+0.j]
    [8.8817842e-16+0.j 1.1358200e+01+0.j 8.8817842e-16+0.j]
    [8.8817842e-16+0.j 8.8817842e-16+0.j 1.1358200e+01+0.j]]
```

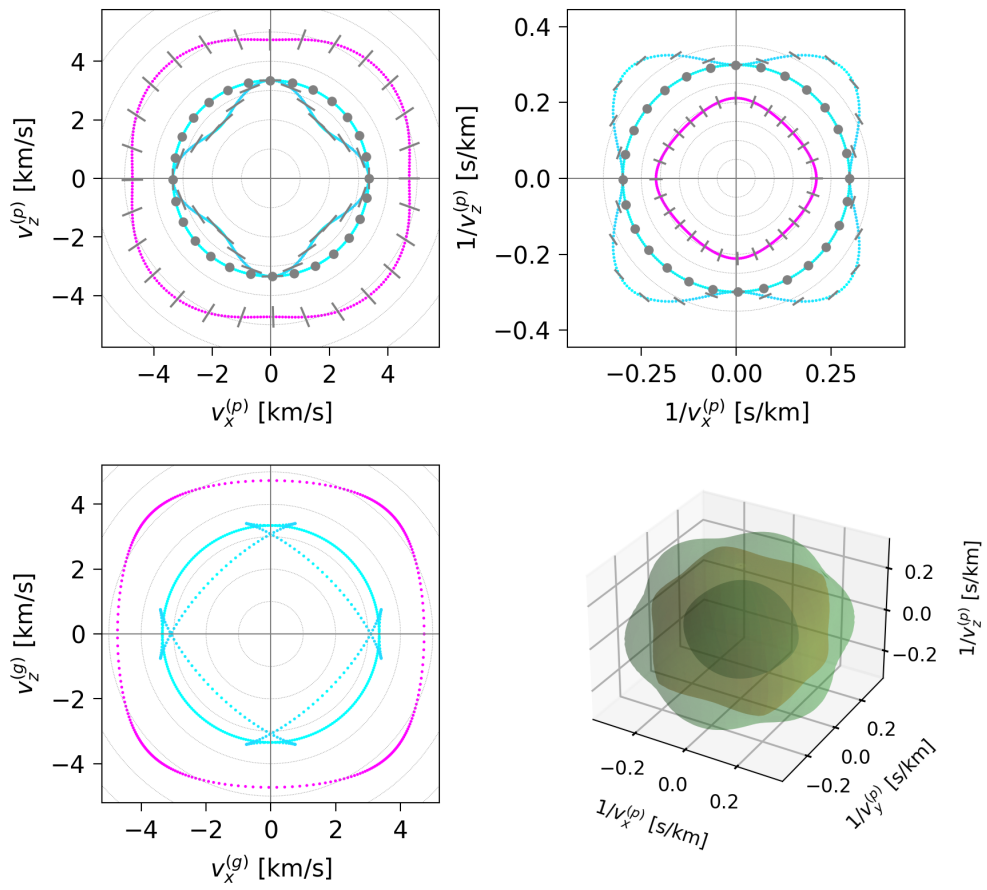
Now the phase and group velocities are different and the eigenstates are hybrid in character with polarisation vectors pointing along irregular directions. Nevertheless, the first mode is close to longitudinal, the second mode is close to shear, and the third is pure shear. This phenomenon of "quasi-longitudinal", "quasi-shear" and pure shear modes is common.

### Dispersion diagrams

We can obtain a much fuller picture by plotting several bulk dispersion properties as a function of the wavevector in 2D and 3D.

Here's the dispersion maps for propagation in the $x$-$z$ plane for the default orientation of GaAs. This plot can be compared with Auld Fig. 7.2.

[50]:
```
prefix = 'tmpgaas'
mat_gaas.plot_bulk_dispersion(prefix)
display(Image(prefix+'-bulkdisp.png', width=600))
```



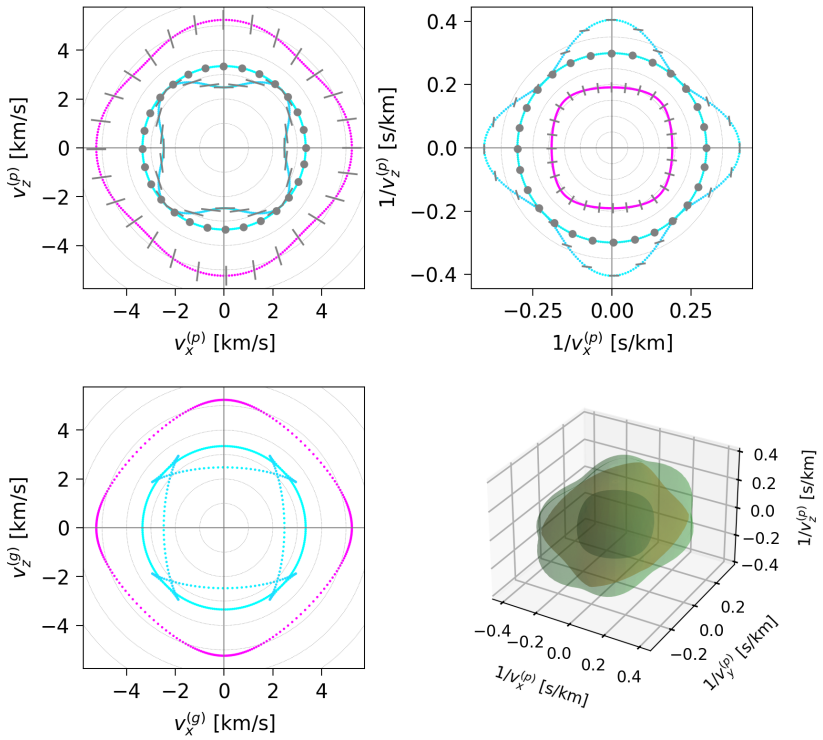

These plots respectively show contours of the *phase velocity* surface (top-left), the *slowness* surface $1/v_p(\vec{\kappa})$ (top-right), the *ray* or group velocity surface (bottom-left) and the full 3D slowness surface (bottom-right).

The colours in the first three plots correspond to the component of each wave mode's elastic polarisation along the propagation direction, ie $r = \hat{\kappa} \cdot \hat{u} = \hat{z} \cdot \hat{u}$. The lines and dots also indicate the polarisation states. It is apparent that the pink coloured mode is close to longitudinal and the blue modes are close to transverse (shear). It turns out that for a given wavevector, the group velocity is *normal* to the slowness surface. Tracing around the outer curve quasi-shear mode in the first plot can help to understand the cusps in the corresponding curve of the group velocity plot.

These plots are always shown in the $x$-$z$ plane. To see other cuts, we can rotate the crystal.

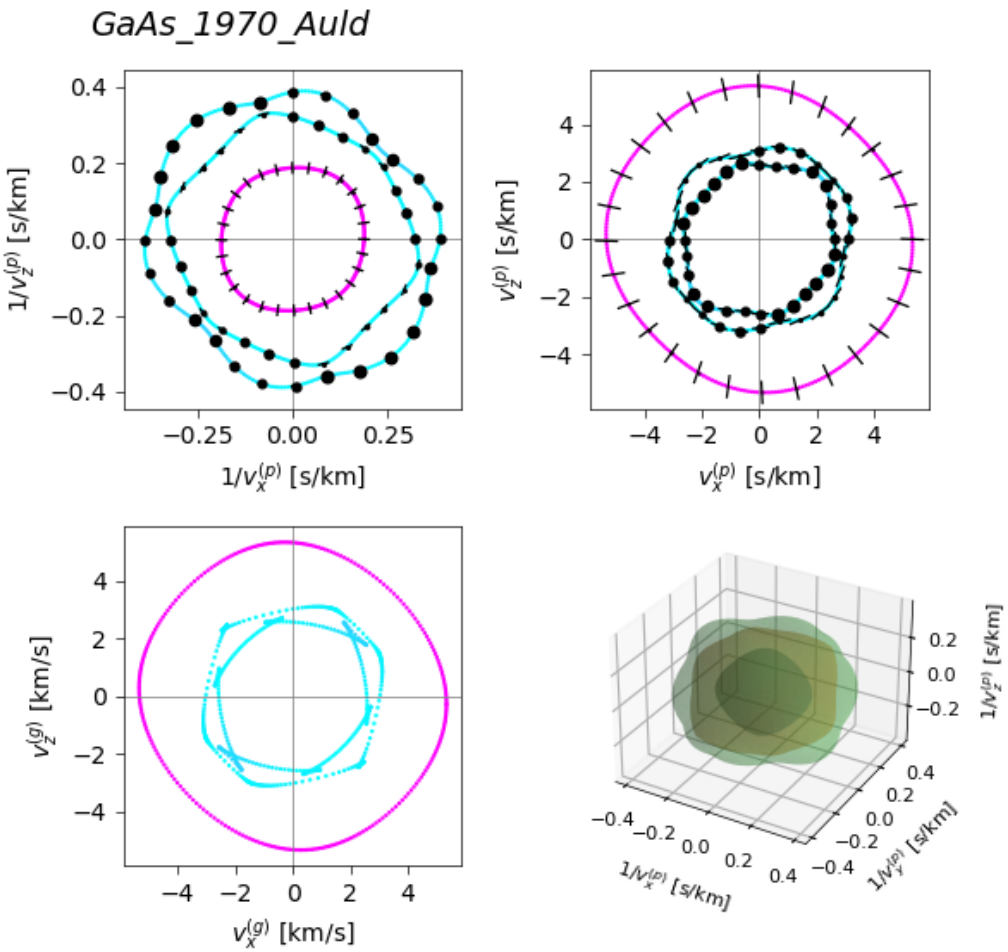
Here is the case for a $\pi/4$ rotation within one of the cubic crystal faces.

[53]:
```
prefix = 'tmpgaas2'
mat_gaas2.plot_bulk_dispersion(prefix)
display(Image(prefix+'-bulkdisp.png', width=500))
```




And here is the case for GaAs with the $\pi/3$ rotation:

[ ]:
```
prefix = 'tmpgaas3'
mat_gaas3.plot_bulk_dispersion(prefix)
display(Image(prefix+'-bulkdisp.png', width=500))
```

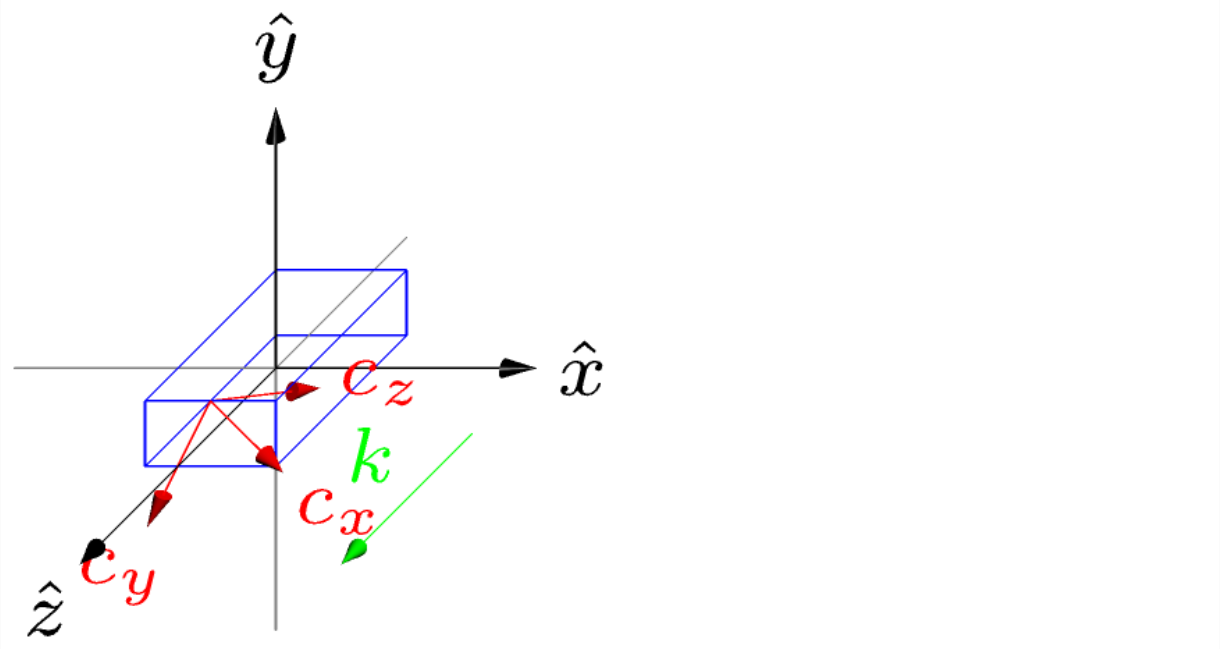



### 7.1.8 Special crystal orientations

Rotations in |NUMBAT| can be specified in several ways.

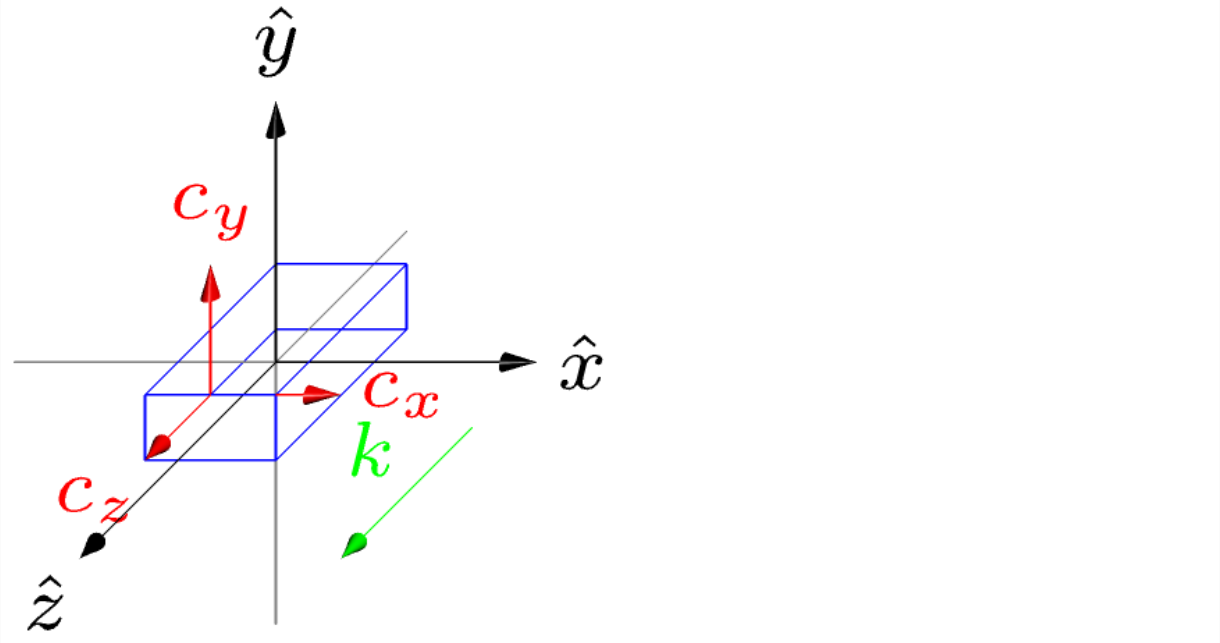
As well as the angle and unit vector, the coordinate axes can be named directly, and rotation calls can be made successively to apply sequences of rotations. Here's a sequence of 3 rotations, rather like a so-called "Euler rotation".

[ ]:
```
mat_3 = mat_a.copy()

mat_3.rotate('x-axis', np.pi/4)         # Apply a positive pi/4 rotation around the␣
↪lab x axis
mat_3.rotate('z', np.pi/5)              # Now apply a positive pi/5 rotation around␣
↪the lab z axis
mat_3.rotate('x-axis', -4*np.pi/3)      # Now apply a negative -4pi/3 rotation around␣
↪the lab x axis

pref='tmp3'
mat_3.make_crystal_axes_plot(pref)
display(Image(pref+'-crystal.png', width=300))
```




To return to the starting configuration, use `reset_orientation()` (or just make a new material from scratch).

[ ]:
```
mat_3.reset_orientation()
mat_3.make_crystal_axes_plot(pref)
display(Image(pref+'-crystal.png', width=300))
```




Some materials define special directions which are commonly encountered. For example, many important semiconductors such as lithium niobate can be obtained in so-called *x-cut*, *y-cut* or *z-cut* varieties.

The orientations corresponding to each of these materials can be obtained by applying sequences of rotations like above. However, it is also possible to define specific rotations in the `.json` file.

For lithium niobate, which has trigonal symmetry, the default orientation is *y-cut*, with the optical symmetry axis $\hat{c}_z$ pointing along the $\hat{z}$ direction:

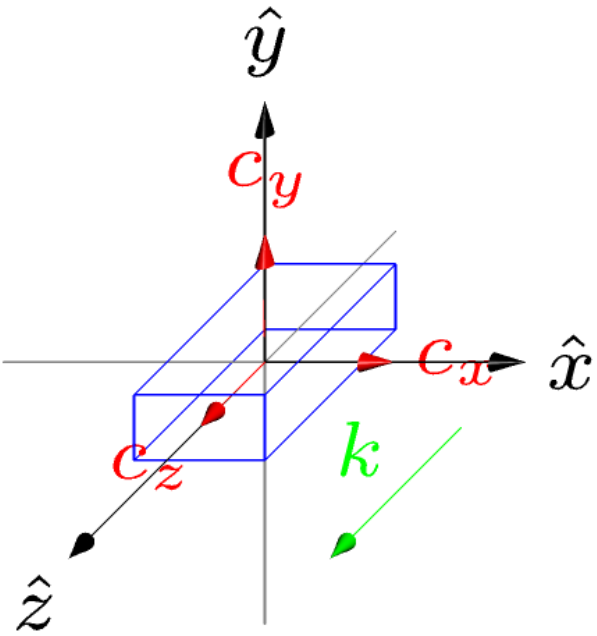
[36]:
```
mat_LiNb_y = materials.make_material('LiNbO3aniso_2021_Steel')
pref='tmp_linb'
```

(continues on next page)

(continued from previous page)

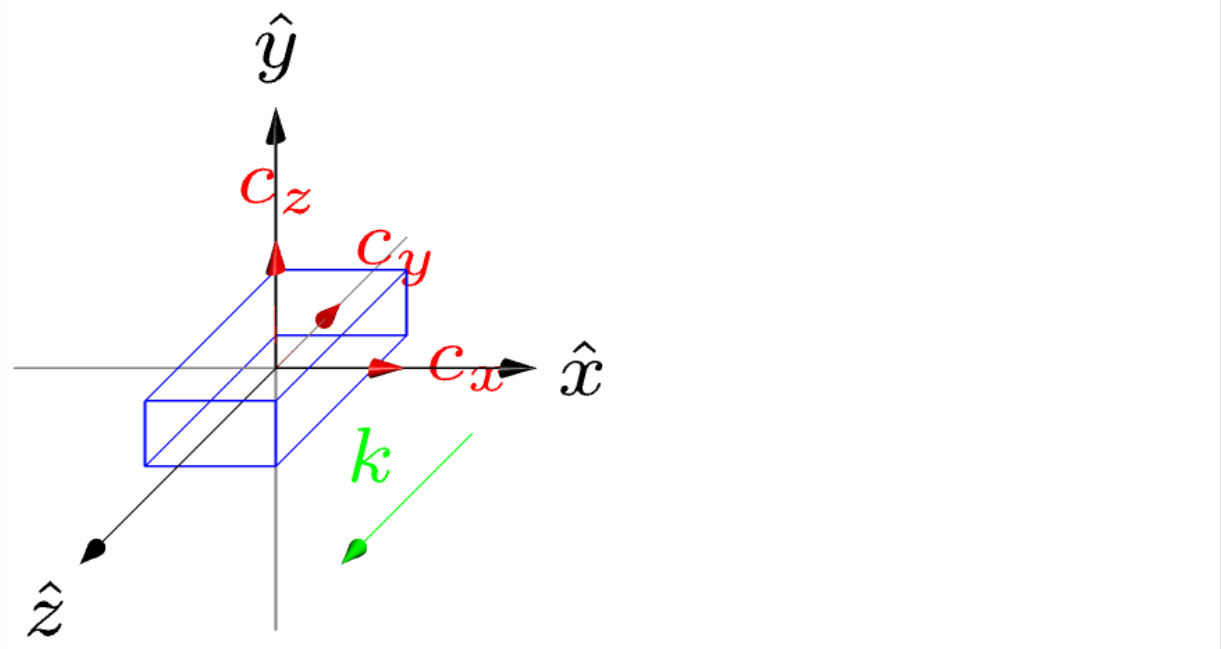
```
mat_LiNb_y.make_crystal_axes_plot(pref+'-ycut')
display(Image(pref+'-ycut-crystal.png', width=300))
```

```
{'piezo_active': 1, 'piezo_d_x5': 6.8e-11, 'piezo_d_y2': 2.1e-11, 'piezo_d_z1': -1e-
↪12, 'piezo_d_z3': 6e-12, 'crystal_sym': '3m'}
{'piezo_active': 1, 'piezo_d_x4': 2.6e-12, 'crystal_sym': "4'3m"}
{'piezo_active': 1, 'piezo_d_x5': -1.4e-11, 'piezo_d_z1': -5e-12, 'piezo_d_z3': 1.03e-
↪11, 'crystal_sym': '6mm'}
{'piezo_active': 0, 'piezo_d_x5': 6.8e-11, 'piezo_d_y2': 2.1e-11, 'piezo_d_z1': -1e-
↪12, 'piezo_d_z3': 6e-12, 'piezo_d_x6': -4.2e-11, 'piezo_d_y1': -2.1e-11, 'piezo_d_y4
↪': 6.8e-11, 'piezo_d_z2': -1e-12, 'crystal_sym': 'no sym'}
```




Instead, selecting the *z-cut* orientation reorients the $\hat{c}_z$ axis to point along $-\hat{y}$ by applying a $\pi/2$ rotation around the positive $\hat{x}$ axis:

[37]:
```
mat_LiNb_z = mat_LiNb_y.copy()
mat_LiNb_z.set_orientation('z-cut')
mat_LiNb_z.make_crystal_axes_plot(pref+'-zcut')
display(Image(pref+'-zcut-crystal.png', width=300))
```




The different direction names here are admittedly confusing, because our choice of coordinates for the waveguide directions is not universal. The key point is that for the $z$-cut orientation, the $\hat{c}_z$ axis points upwards out of the waveguide which is the standard convention.

The different orientations are reflected in the bulk dispersion properties of the different cases as shown below (we add in the $y$-cut case too). The 3D plots are identical, but the projections in the $x$-$z$ plane are different. The $z$-cut case shows the full 6-fold symmetry of the hexagonal crystal. The orientation of the $x$-cut case leads to a 4-fold symmetry for propagation in the $x$-$z$ plane.

[38]:
```
print(mat_LiNb_y.elastic_properties())
print(mat_LiNb_z.elastic_properties())
```

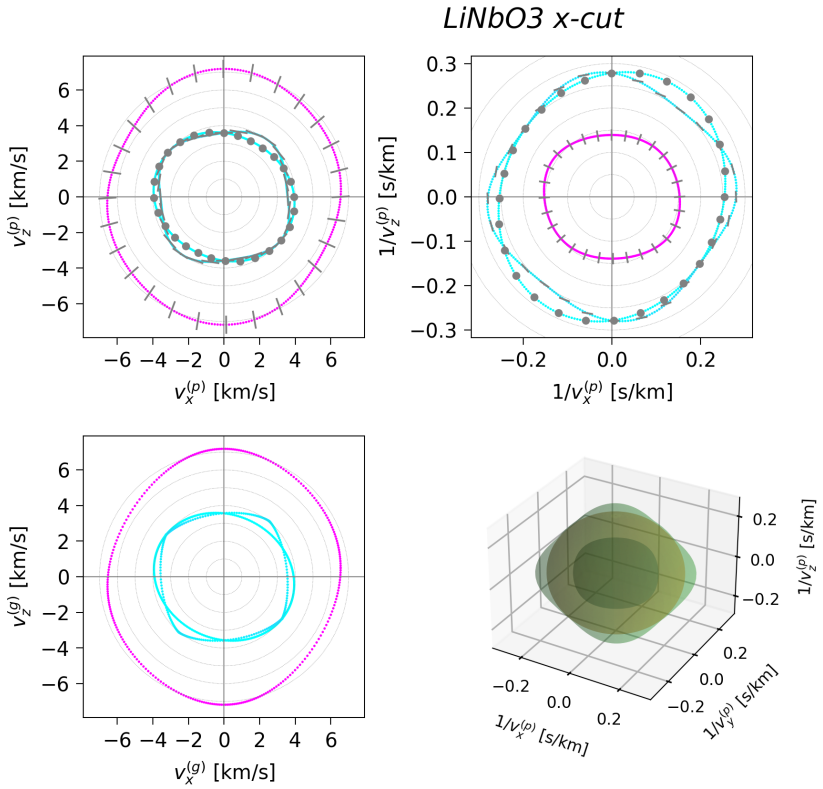
```
Elastic properties of material LiNbO3aniso_2021_Steel
  Density:         4650.000 kg/m^3
  Ref. index:      2.3000+0.0000j
  Crystal class:   Trigonal
  Crystal group:   no sym
  Stiffness c_IJ:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[199.2   54.7   70.     7.9    0.     0.  ]
     [ 54.7  199.2   70.    -7.9    0.     0.  ]
     [ 70.    70.   240.     0.     0.     0.  ]
     [  7.9   -7.9    0.    59.9    0.     0.  ]
     [  0.     0.     0.     0.    59.9    7.9 ]
     [  0.     0.     0.     0.     7.9   72.25]]

  Wave mode 1: v_p=7.1842 km/s,  |v_g|=7.1842 km/s,  u_j=[ 0.0000  0.0000  1.0000], ␣
↪v_g=[ 0.0000  0.0000  7.1842] km/s
  Wave mode 2: v_p=3.5891 km/s,  |v_g|=3.6202 km/s,  u_j=[ 1.0000  0.0000  0.0000], ␣
↪v_g=[ 0.0000  0.4734  3.5891] km/s
  Wave mode 3: v_p=3.5891 km/s,  |v_g|=3.6202 km/s,  u_j=[ 0.0000  1.0000  0.0000], ␣
↪v_g=[ 0.0000 -0.4734  3.5891] km/s
Piezoelectric properties:
 Piezo effects disabled

 Strain piezo coefficient d_iJ, unit: pC/N.
```

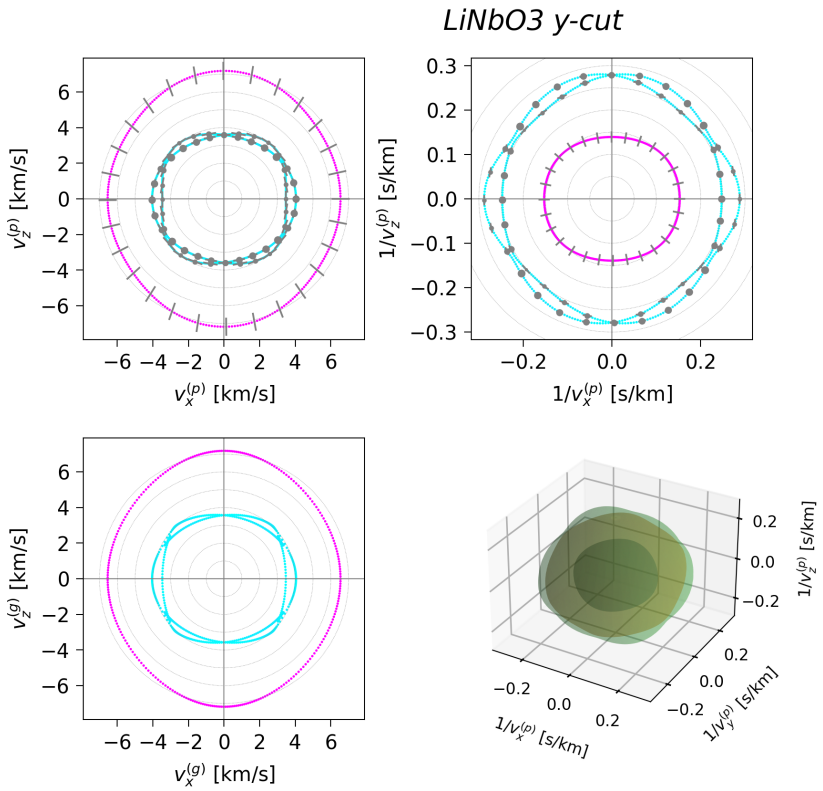
(continues on next page)

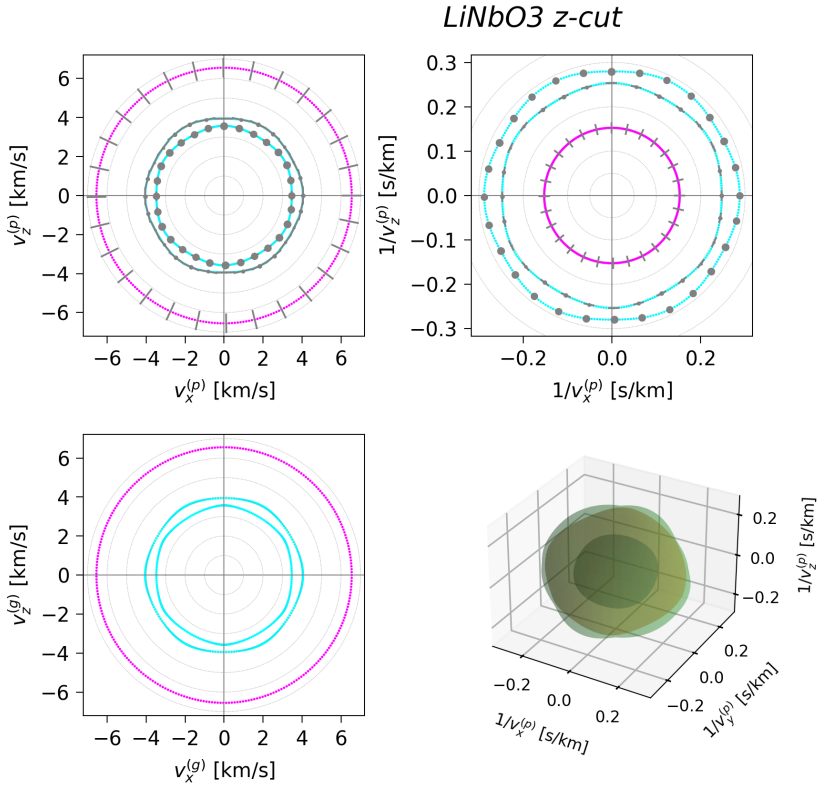
(continued from previous page)

```
   Voigt 3-tensor:
    [[  0.   0.   0.   0.  68. -42.]
     [-21.  21.   0.  68.   0.   0.]
     [ -1.  -1.   6.   0.   0.   0.]]

 Stress piezo coefficient e_iJ, unit: C/m^2.
   Voigt 3-tensor:
    [[ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  3.7414e+00 -2.4973e+00]
     [-2.4973e+00  2.4973e+00 -7.3603e-17  3.7414e+00  0.0000e+00  0.0000e+00]
     [ 1.6610e-01  1.6610e-01  1.3000e+00  8.0422e-19  0.0000e+00  0.0000e+00]]

 Dielectric tensor epsS_ij
    [[ 5.29000000e+00+0.j  0.00000000e+00+0.j  0.00000000e+00+0.j]
     [ 0.00000000e+00+0.j  5.29000000e+00+0.j  1.00025989e-27+0.j]
     [ 0.00000000e+00+0.j -1.20289838e-28+0.j  5.29000000e+00+0.j]]
Elastic properties of material LiNbO3aniso_2021_Steel
  Density:        4650.000 kg/m^3
  Ref. index:     2.3000+0.0000j
  Crystal class:  Trigonal
  Crystal group:  no sym
  Stiffness c_IJ:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[ 1.9920e+02  7.0000e+01  5.4700e+01 -7.9000e+00  0.0000e+00  0.0000e+00]
     [ 7.0000e+01  2.4000e+02  7.0000e+01  3.0739e-15  0.0000e+00  0.0000e+00]
     [ 5.4700e+01  7.0000e+01  1.9920e+02  7.9000e+00  0.0000e+00  0.0000e+00]
     [-7.9000e+00  3.0739e-15  7.9000e+00  5.9900e+01  0.0000e+00  0.0000e+00]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  7.2250e+01 -7.9000e+00]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00 -7.9000e+00  5.9900e+01]]

  Wave mode 1: v_p=6.5525 km/s,  |v_g|=6.5714 km/s,  u_j=[ 0.0000  0.0564  0.9984], ␣
↪v_g=[ 0.0000  0.4987  6.5525] km/s
  Wave mode 2: v_p=3.9418 km/s,  |v_g|=3.9653 km/s,  u_j=[ 1.0000  0.0000  0.0000], ␣
↪v_g=[ 0.0000 -0.4310  3.9418] km/s
  Wave mode 3: v_p=3.5757 km/s,  |v_g|=3.6025 km/s,  u_j=[ 0.0000  0.9984 -0.0564], ␣
↪v_g=[ 0.0000 -0.4387  3.5757] km/s
Piezoelectric properties:
 Piezo effects disabled

 Strain piezo coefficient d_iJ, unit: pC/N.
   Voigt 3-tensor:
    [[ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  4.1638e-15]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  4.1638e-15]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00 -6.8000e+01]]

 Stress piezo coefficient e_iJ, unit: C/m^2.
   Voigt 3-tensor:
    [[ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  2.2909e-16]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  2.2990e-16]
     [ 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00 -3.7414e+00]]

 Dielectric tensor epsS_ij
    [[5.29000000e+00+0.j 0.00000000e+00+0.j 0.00000000e+00+0.j]
     [0.00000000e+00+0.j 5.29000000e+00+0.j 2.16639447e-26+0.j]
     [0.00000000e+00+0.j 2.05433678e-26+0.j 5.29000000e+00+0.j]]
```

[39]:
```
mat_LiNb_x = mat_LiNb_y.copy()
mat_LiNb_x.set_orientation('x-cut')

mat_LiNb_x.plot_bulk_dispersion(pref+'-xcut', label='LiNbO3 x-cut')
mat_LiNb_y.plot_bulk_dispersion(pref+'-ycut', label='LiNbO3 y-cut')
mat_LiNb_z.plot_bulk_dispersion(pref+'-zcut', label='LiNbO3 z-cut')

display(Image(pref+'-xcut-bulkdisp.png', width=500))
display(Image(pref+'-ycut-bulkdisp.png', width=500))
display(Image(pref+'-zcut-bulkdisp.png', width=500))
```




LiNbO3 y-cut
$v_z^{(p)}$ [km/s]
$v_x^{(p)}$ [km/s]
$1/v_z^{(p)}$ [s/km]
$1/v_x^{(p)}$ [s/km]
$v_z^{(g)}$ [km/s]
$v_x^{(g)}$ [km/s]
$1/v_x^{(p)}$ [s/km]
$1/v_y^{(p)}$ [s/km]
$1/v_z^{(p)}$ [s/km]


LiNbO3 z-cut
$v_z^{(p)}$ [km/s]
$v_x^{(p)}$ [km/s]
$1/v_z^{(p)}$ [s/km]
$1/v_x^{(p)}$ [s/km]
$v_z^{(g)}$ [km/s]
$v_x^{(g)}$ [km/s]
$1/v_x^{(p)}$ [s/km]
$1/v_y^{(p)}$ [s/km]
$1/v_z^{(p)}$ [s/km]

### 7.1.9 Bulk dispersion and core-cladding guidance

A useful application of the bulk dispersion curves is as a tool to predict the guidance properties of two media by comparing their slowness curves.

Consider the first non-fibre conventional waveguide to show SBS: a chalcogenide ($As_2S_3$) strip waveguide on a silica substrate.

[45]:
```
mat_SiO2 = materials.make_material('SiO2_2021_Poulton')
mat_As2S3 = materials.make_material('As2S3_2021_Poulton')

print(mat_SiO2.elastic_properties(), '\n\n')

print(mat_As2S3.elastic_properties())
```

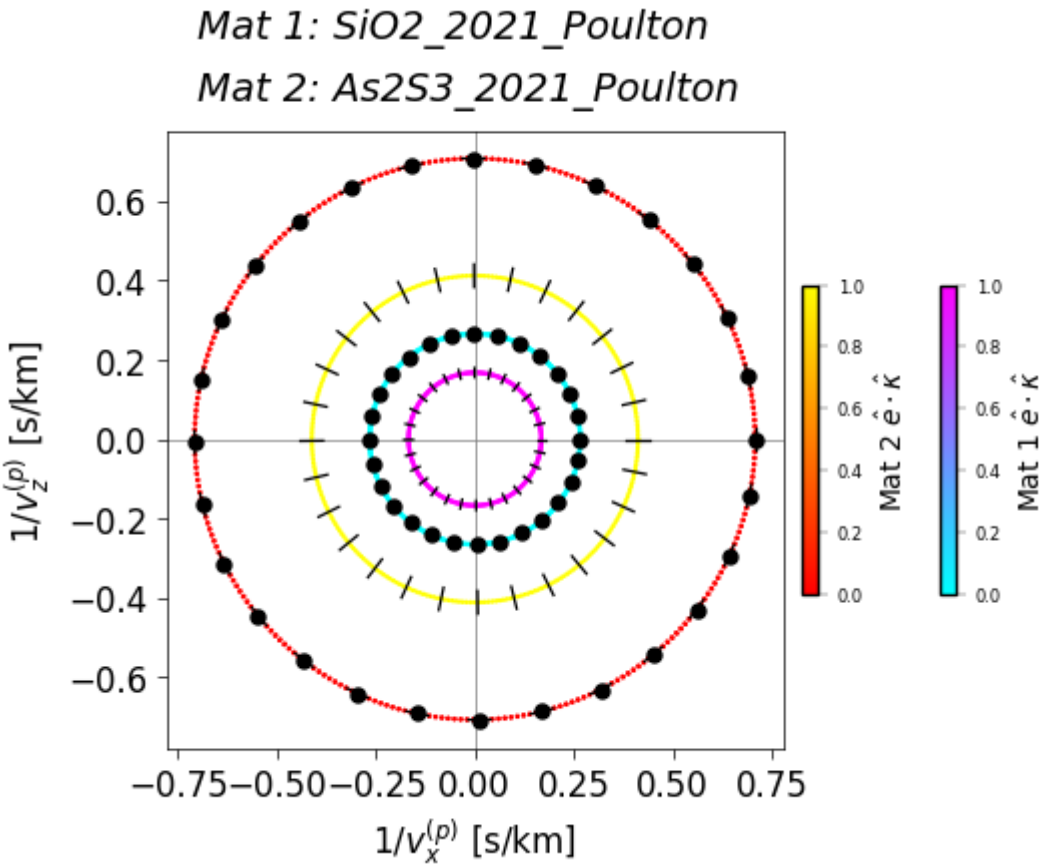
```
Elastic properties of material SiO2_2021_Poulton
  Density:        2200.000 kg/m^3
  Ref. index:     1.4500+0.0000j
  Crystal class:  Isotropic
  Crystal group:  no sym
  c11:            78.500 GPa
  c12:            16.100 GPa
  c44:            31.200 GPa
  Young's mod E:  73.020 GPa
  Poisson ratio:  0.170
  Velocity long.: 5973.426 m/s
  Velocity shear: 3765.875 m/s
  Velocity Rayleigh: 3411.154 m/s


Elastic properties of material As2S3_2021_Poulton
  Density:        3200.000 kg/m^3
  Ref. index:     2.4500+0.0000j
  Crystal class:  Isotropic
  Crystal group:  no sym
  c11:            18.900 GPa
  c12:            6.000 GPa
  c44:            6.400 GPa
  Young's mod E:  15.897 GPa
  Poisson ratio:  0.242
  Velocity long.: 2430.278 m/s
  Velocity shear: 1414.214 m/s
  Velocity Rayleigh: 1298.833 m/s
```

Noting that the chalcogenide refractive index is higher, we are motivated by optical guidance to use it as the core material.

We then note that both the elastic wave velocities for the chalcogenide are lower than the shear velocity for the silica. Consequently, we can expect the chalcogenide to form a suitable elastic cladding, which is indeed the case and explains why this system successfully showed SBS in 2012.

For isotropic materials, this is sufficient investigation, but we can confirm the result by comparing the slowness curves for both materials on one plot:

[46]:
```
materials.compare_bulk_dispersion(mat_SiO2, mat_As2S3, 'comp_sio2_as2s3')
```




The slowness curves for silica are shown in red/orange, those for the chalcogenide are shown in blue and magenta. Since the latter are entirely contained in the former, the chalcogenide is an elastically slow material and forms an ideal cladding.

Now consider the silicon/silica or SOI system.

[47]:
```
mat_SiO2 = materials.make_material('SiO2_2021_Poulton')
mat_Si = materials.make_material('Si_1970_Auld')

print(mat_SiO2.elastic_properties(), '\n\n')

print(mat_Si.elastic_properties())
```

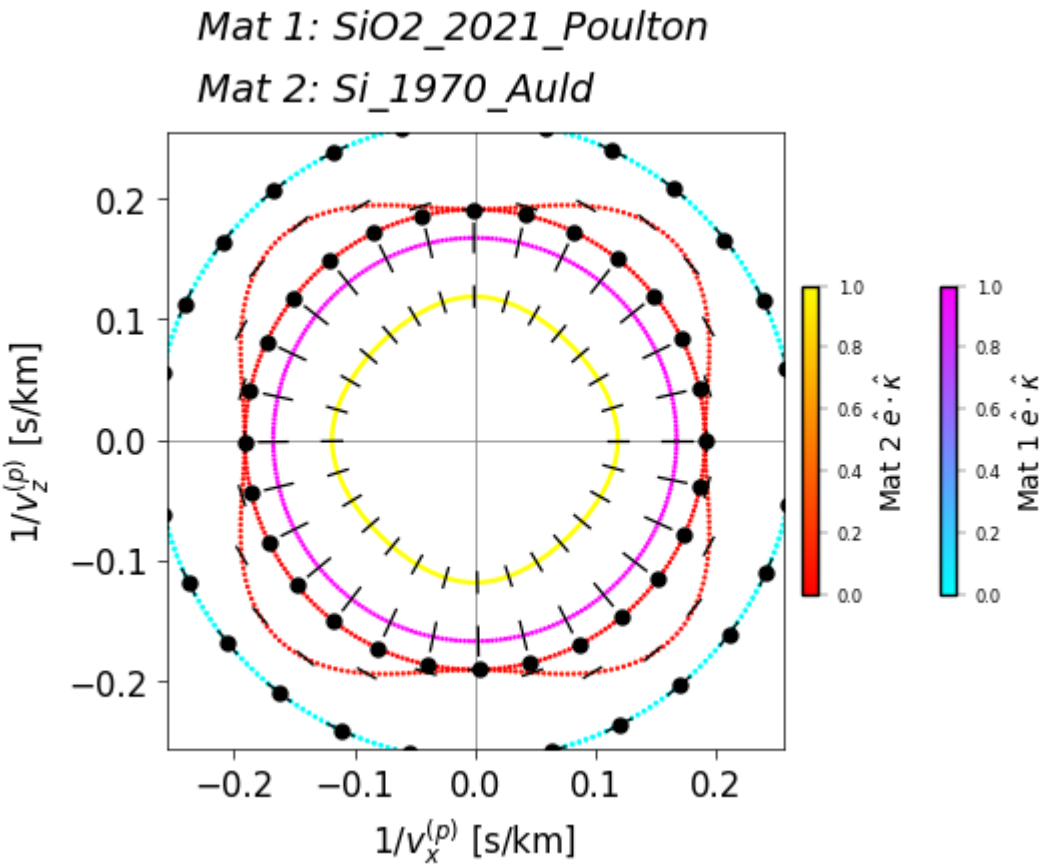
```
Elastic properties of material SiO2_2021_Poulton
  Density:         2200.000 kg/m^3
  Ref. index:      1.4500+0.0000j
  Crystal class:   Isotropic
  Crystal group:   no sym
  c11:             78.500 GPa
  c12:             16.100 GPa
  c44:             31.200 GPa
  Young's mod E:   73.020 GPa
  Poisson ratio:   0.170
  Velocity long.: 5973.426 m/s
  Velocity shear: 3765.875 m/s
  Velocity Rayleigh: 3411.154 m/s


Elastic properties of material Si_1970_Auld
```

(continues on next page)

(continued from previous page)

```
 Density:         2332.000 kg/m^3
 Ref. index:      3.5000+0.0000j
 Crystal class:   Cubic
 Crystal group:   no sym
 Stiffness c_IJ:
Stiffness c_IJ, unit: GPa.
  Voigt 4-tensor:
   [[165.7   79.56  79.56   0.     0.     0.  ]
    [ 79.56 165.7   79.56   0.     0.     0.  ]
    [ 79.56  79.56 165.7    0.     0.     0.  ]
    [  0.     0.     0.    63.9    0.     0.  ]
    [  0.     0.     0.     0.    63.9    0.  ]
    [  0.     0.     0.     0.     0.    63.9 ]]

 Wave mode 1: v_p=8.4294 km/s,  |v_g|=8.4294 km/s,  u_j=[ 0.0000  0.0000  1.0000], ␣
↪v_g=[ 0.0000  0.0000  8.4294] km/s
 Wave mode 2: v_p=5.2346 km/s,  |v_g|=5.2346 km/s,  u_j=[ 1.0000  0.0000  0.0000], ␣
↪v_g=[ 0.0000  0.0000  5.2346] km/s
 Wave mode 3: v_p=5.2346 km/s,  |v_g|=5.2346 km/s,  u_j=[ 0.0000  1.0000  0.0000], ␣
↪v_g=[ 0.0000  0.0000  5.2346] km/s
```

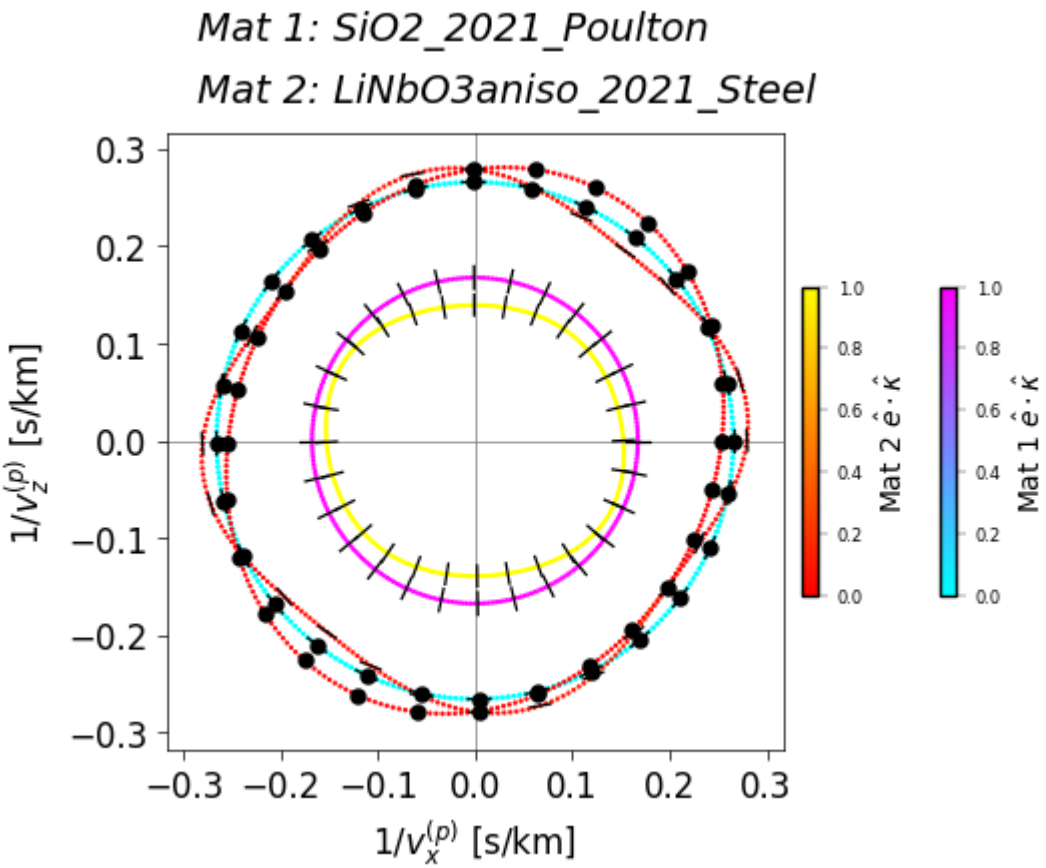
[48]:
```
materials.compare_bulk_dispersion(mat_SiO2, mat_Si, 'comp_sio2_si')
```

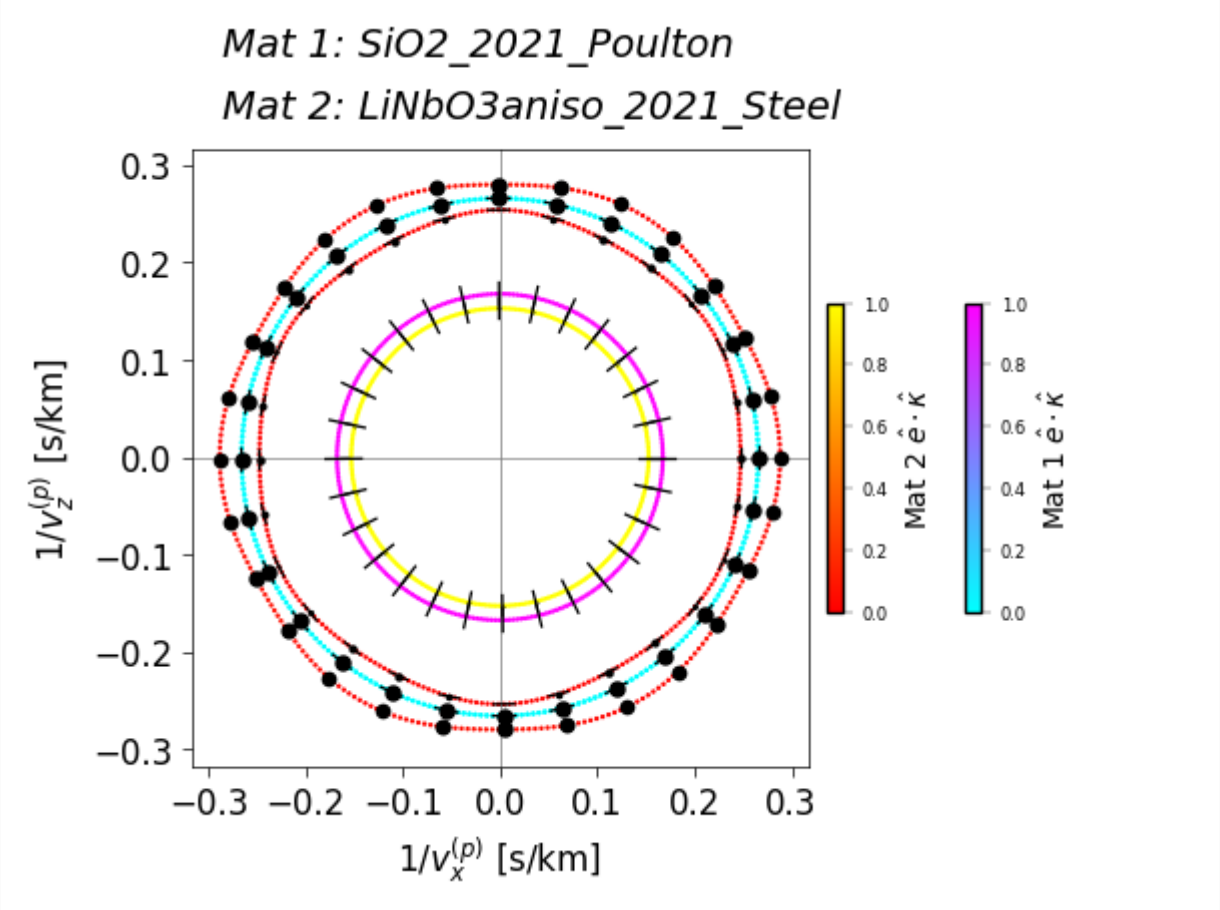



Here we see that the slowness curves are interleaved but both families of waves are slower in silica than their corresponding modes in silicon. Consequently, we can't guide both sound and light in a conventional SOI waveguide, and all SBS demonstrations in this class of platform have involved special techniques such as undercut waveguides or pillar structures.

A similar case arises with lithium niobate and silica (and a number of other potential substrates). Lithium niobate and silica form an excellent core-cladding combination for light guidance but the elastic wave situation is as follows:

[49]:
```
materials.compare_bulk_dispersion(mat_SiO2, mat_LiNb_x, 'comp_sio2_linb_x')

materials.compare_bulk_dispersion(mat_SiO2, mat_LiNb_z, 'comp_sio2_linb_z')
```






This material combination fails for both common crystal orientations.

However, as several groups have realised, while these elastic properties forbid total internal reflection elastic guidance in a conventional waveguide, and it does not forbid efficient elastic guidance as a Rayleigh-like surface mode.

## 7.2 Introduction to piezoelectric properties with lithium niobate

Continuing the study of bulk material elastic properties from the previous tutorial, let's take a look at the orientational dependence of lithium niobate, an increasingly important material in SBS.

The numerical values for the $LiNbO_3$ material properties are taken from Rodrigues et al, JOSA B 40, D56 (2023).

[ ]:
```
%load_ext autoreload
%autoreload 3

import sys
#import os
import time
import numpy as np
from IPython.display import Image, display, HTML

sys.path.append("../backend")
import materials

def img_single(fn, cap='', width=300):
    cache_buster=int(time.time() * 1000)
    #<div style="display: flex; justify-content: center; align-items: flex-start;">
    #</div>
```

(continues on next page)

(continued from previous page)

```
    html_string = f"""
        <div style="margin-right: 10px;">
            <img src="{fn}?{cache_buster}" width="{width}px">
            <p style="text-align: center;">{cap}</p>
        </div>"""

    display(HTML(html_string))

def img_pair(fn1, fn2, cap1='', cap2='', width1=300, width2=300):
    cache_buster=int(time.time() * 1000)
    html_string = f"""
    <div style="display: flex; justify-content: center; align-items: flex-start;">
        <div style="margin-right: 10px;">
            <img src="{fn1}?{cache_buster}" width="{width1}px">
            <p style="text-align: center;">{cap1}</p>
        </div>
        <div>
            <img src="{fn2}?{cache_buster}" width="{width2}px">
            <p style="text-align: center;">{cap2}</p>
        </div>
    </div>"""

    display(HTML(html_string))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

### 7.2.1 Stiffness tensor under crystal rotations

#### The default orientation: $y$-cut

To review, in NumBAT, the *laboratory* axes $x$, $y$ and $z$ are fixed. Propagation always occurs along the $z$ direction and the vertical direction out of the substrate is always along $y$.

The *crystal* axes of a material, denoted $X$, $Y$ and $Z$, or $\hat{c}_x$, $\hat{c}_y$, $\hat{c}_z$ can be rotated.

The material is defined in the definition json files with the crystal and laboratory axes aligned, and is thus in $y$-cut form: the crystal $\hat{c}_y$ axis points up out of the waveguide, with the crystal symmetry axis $\hat{c}_z$ lying along the propagation direction $z$.

We'll look at this for a material file describing lithium niobate. (Remember that NumBAT, the same material can have multiple definition files to allow exact comparison with literature references.)

[196]:
```
mat_LiNbO3 = materials.make_material("LiNbO3_2023_Rodrigues")
mat_LiNbO3_y = mat_LiNbO3  # for naming convenience later
```

Here is the default orientation as just described with the crystal and laboratory axes aligned.

[197]:
```
pref='tmp_LiNbO3'
fnimg = mat_LiNbO3.make_crystal_axes_plot(pref+'-ycut')
img_single(fnimg, width=300)
```

```
<IPython.core.display.HTML object>
```

The crystal properties show that for this orientation the two shear modes for propagation along $z$ are degenerate:

[198]:
```
print(mat_LiNbO3.elastic_properties())
```

```
Elastic properties of material LiNbO3_2023_Rodrigues
  Density:        4650.000 kg/m^3
  Ref. index:     2.2100+0.0000j
  Crystal class:  Trigonal
  Crystal group:  no sym
  Stiffness c_IJ:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[198.8300  54.6400  68.2300   7.8300   0.0000   0.0000]
     [ 54.6400 198.8300  68.2300  -7.8300   0.0000   0.0000]
     [ 68.2300  68.2300 235.7100   0.0000   0.0000   0.0000]
     [  7.8300  -7.8300   0.0000  59.8600   0.0000   0.0000]
     [  0.0000   0.0000   0.0000   0.0000  59.8600   7.8300]
     [  0.0000   0.0000   0.0000   0.0000   7.8300  72.0950]]

  Wave mode 1: v_p=7.1197 km/s,  |v_g|=7.1197 km/s,  u_j=[ 0.0000  0.0000  1.0000], ␣
↪v_g=[ 0.0000  0.0000  7.1197] km/s
  Wave mode 2: v_p=3.5879 km/s,  |v_g|=3.6185 km/s,  u_j=[ 1.0000  0.0000  0.0000], ␣
↪v_g=[ 0.0000  0.4693  3.5879] km/s
  Wave mode 3: v_p=3.5879 km/s,  |v_g|=3.6185 km/s,  u_j=[ 0.0000  1.0000  0.0000], ␣
↪v_g=[ 0.0000 -0.4693  3.5879] km/s
```

This is not true if the wave vector direction in the $x$-$z$ plane is varied.

By solving the Christoffel equation as the wave vector rotates in the $x$-$z$ plane we can map out curves of the phase velocity, inverse phase velocity (or "slowness"), and the magnitude of the group velocity $|v_g(\kappa)|$.

The faint circular lines mark radial velocities at 1 km/s intervals.

[199]:
```
imgdisp_y = mat_LiNbO3.plot_bulk_dispersion(pref+'-ycut')
img_single(imgdisp_y, width=400)
```

```
<IPython.core.display.HTML object>
```

For this material, it is quite hard to see the difference between the phase and group velocity plots. That is more obvious in some other materials.

These plots can also be generated without the polarisation state markers:

[200]:
```
imgdisp_yb = mat_LiNbO3.plot_bulk_dispersion(pref, show_poln=False)
img_single(imgdisp_yb, width=400)
```

```
<IPython.core.display.HTML object>
```

### The $z$-cut orientation

Now let's rotate the starting crystal so the primary crystal symmetry axis $\hat{c}_z$ points upwards along $\hat{y}$, with the $\hat{c}_y$ axis along $-\hat{z}$. This corresponds to Fig. 2a in Rodrigues et al.

[201]:
```
mat_LiNbO3_z = mat_LiNbO3.copy()
mat_LiNbO3_z.set_orientation('z-cut')
```

[202]:
```
fimg = mat_LiNbO3_z.make_crystal_axes_plot(pref+'-zcut')
img_single(fimg, width=300)
```

```
<IPython.core.display.HTML object>
```

With the 6-fold symmetry axis $Z$ pointing up along $y$, the dispersion cuts in the $x$-$z$ plane now display the full 6-fold symmetry associated with the trigonal structure of lithium niobate:

[203]:
```
imgdisp_z = mat_LiNbO3_z.plot_bulk_dispersion(pref+'-zcut')
img_single(imgdisp_z,width=500)
```

```
<IPython.core.display.HTML object>
```

### The $x$-cut orientation

For completeness, we can look at the $x$-cut properties:

[204]:
```
mat_LiNbO3_x =  mat_LiNbO3.copy()
mat_LiNbO3_x.set_orientation('x-cut')

print(mat_LiNbO3_x.elastic_properties())
```

```
Elastic properties of material LiNbO3_2023_Rodrigues
  Density:        4650.000 kg/m^3
  Ref. index:     2.2100+0.0000j
  Crystal class:  Trigonal
  Crystal group:  no sym
  Stiffness c_IJ:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[198.8300  54.6400  68.2300  -0.0000   7.8300   0.0000]
     [ 54.6400 198.8300  68.2300   0.0000  -7.8300  -0.0000]
     [ 68.2300  68.2300 235.7100   0.0000   0.0000   0.0000]
     [ -0.0000   0.0000   0.0000  59.8600   0.0000  -7.8300]
     [  7.8300  -7.8300   0.0000   0.0000  59.8600  -0.0000]
     [  0.0000  -0.0000   0.0000  -7.8300  -0.0000  72.0950]]

  Wave mode 1: v_p=7.1197 km/s,  |v_g|=7.1197 km/s,  u_j=[ 0.0000  0.0000  1.0000], ␣
↪v_g=[ 0.0000  0.0000  7.1197] km/s
  Wave mode 2: v_p=3.5879 km/s,  |v_g|=3.6185 km/s,  u_j=[ 1.0000  0.0000  0.0000], ␣
↪v_g=[ 0.4693 -0.0000  3.5879] km/s
  Wave mode 3: v_p=3.5879 km/s,  |v_g|=3.6185 km/s,  u_j=[ 0.0000  1.0000  0.0000], ␣
↪v_g=[-0.4693  0.0000  3.5879] km/s
```

[205]:
```
fnimg = mat_LiNbO3_x.make_crystal_axes_plot(pref+'-xcut')
img_single(fnimg,width=300)
```

```
<IPython.core.display.HTML object>
```

The distinction between the $x$-cut and $y$-cut dispersion is subtle, and it helpful to plot them side by side.

[206]:
```
imgdisp_x = mat_LiNbO3_x.plot_bulk_dispersion(pref+'-xcut')
img_pair(imgdisp_y, imgdisp_x, 'Y cut', 'X cut', 400, 400)
```

```
<IPython.core.display.HTML object>
```

## 7.2.2 Photoelastic response

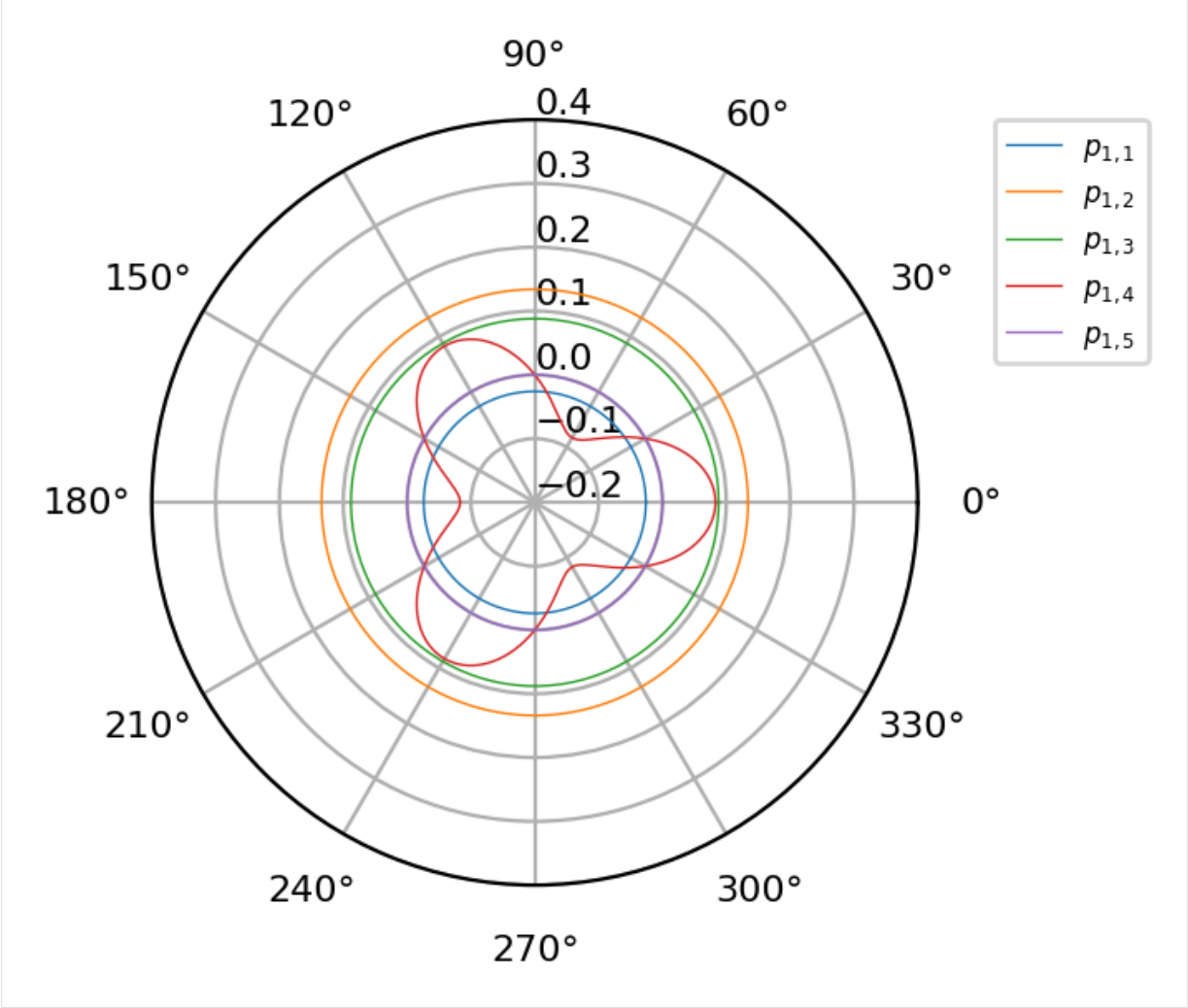
Now let's consider the influence of crystal orientation on the photoelastic coupling.

The photoelastic tensor elements $p_{ijkl}$ also change in value as the crystal is rotated. Once again, we can plot the coefficients in the lab frame coordinates, as a crystal of given orientation is imagined to be rotated around the $y$ axis.

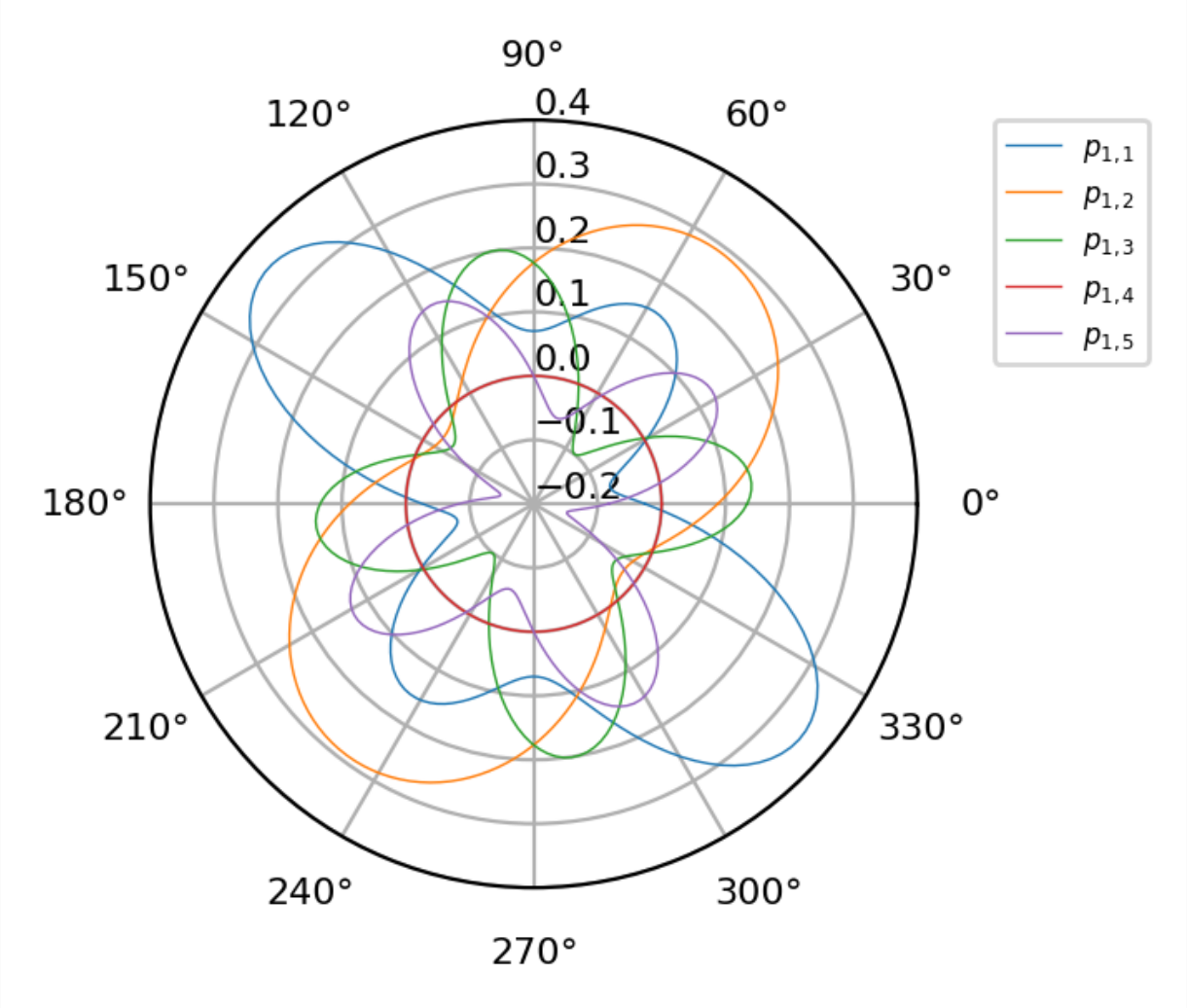
The desired elements of the tensor are specified in the Voigt notation $p_{IJ}$.

As with the results above, the $Z$-cut crystal exhibits the 3-fold trigonal symmetry, while the other two cuts have more complex behaviour.

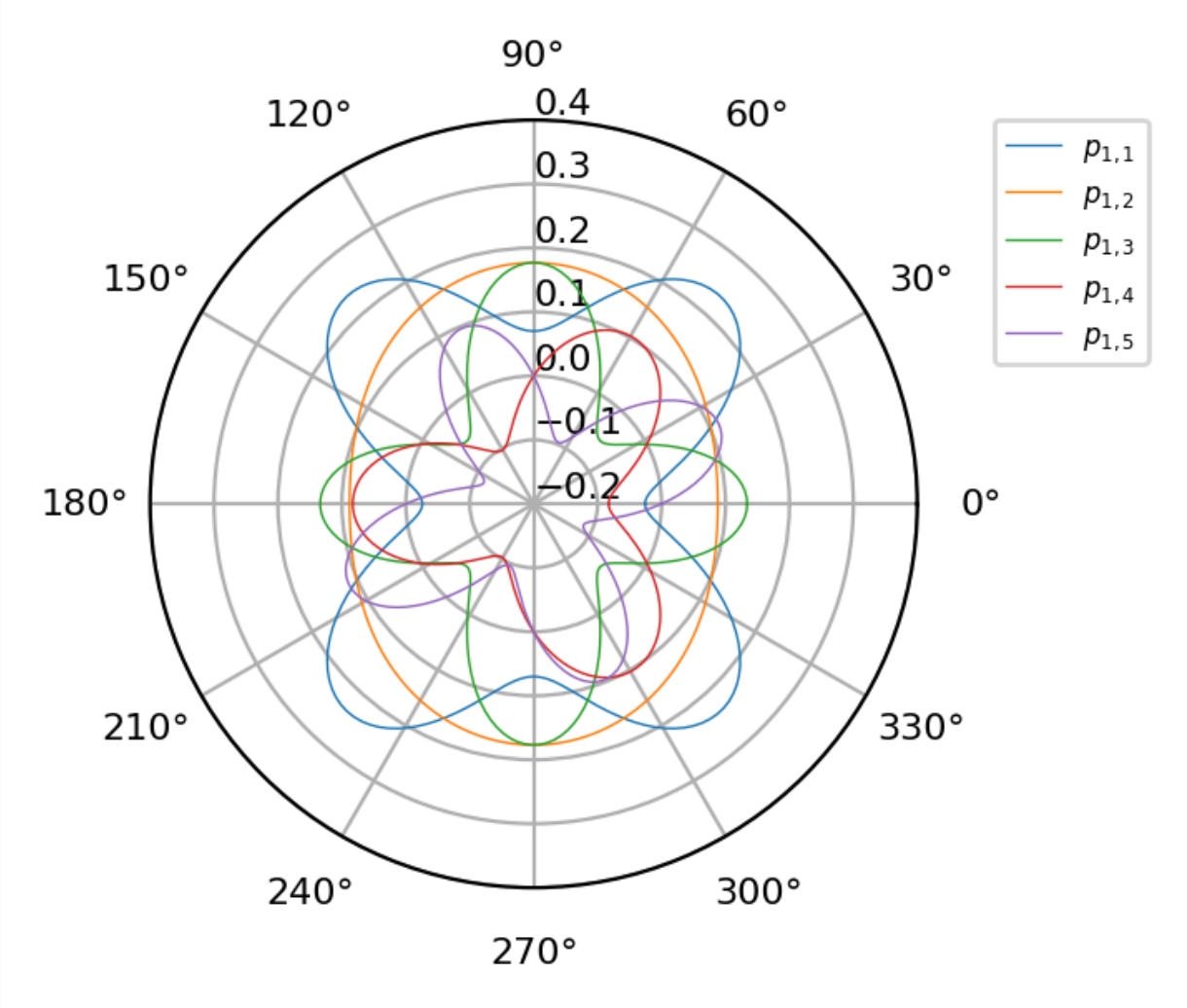
[50]:
```
mat_LiNbO3_z.plot_photoelastic_IJ(pref, ("11","12", "13", "14", "15"))
```




[51]:
```
mat_LiNbO3_x.plot_photoelastic_IJ(pref, ("11","12", "13", "14", "15"))
```




[52]:
```
mat_LiNbO3_y.plot_photoelastic_IJ(pref, ("11","12", "13", "14", "15"))
```




## 7.3 Piezoelectric properties of lithium niobate

Now let's turn to the piezoelectric response of lithium niobate.

This version of lithium niobate has additional piezoelectric quantities defined: the strain piezo coefficient $d_{iJ}$ and the stress piezo coefficient $e_{iJ}$. These are $3 \times 6$ tensors with mixed ordinary and Voigt indices.

### 7.3.1 Piezo slowness curves for x-cut

There are a number of piezoelectric phenomena that involve coupling of the optical field and a material-related polarisation field. Perhaps the most basic effect is a "piezo-stiffening" of the stiffness tensor (a genuinely confusing terminology). This effect is discussed in Auld volume 1, section 8.F.

To good approximation this is an adjustment to the optical properties only in which the "raw" stiffness tensor $c_{IJ}$ is modified by the piezo influence.

It is described by a change to the Christoffel equation so that the bulk optical modes are altered. The revised Christoffel equation has the form

$$q^2 \Gamma_{ij} v_j = \rho \Omega^2 v_i,$$

where

$$\Gamma_{ij} = l_{iK} c_{KL} l_{Lj},$$

and the matrix $l_{Lj}$ indexed by Voigt ($L$) and Cartesian indices ($j$) is composed of the wavevector components $:nbsphinx-math:*vec* `:nbsphinx-math:*kappa* `$:

$$l_{Lj} = \begin{bmatrix} \kappa_x & 0 & 0 \\ 0 & \kappa_y & 0 \\ 0 & 0 & \kappa_z \\ 0 & \kappa_z & \kappa_y \\ \kappa_z & 0 & \kappa_x \\ \kappa_y & \kappa_x & 0 \end{bmatrix}.$$

The *piezoelastically stiffened elastic constant* is

$$c^p_{KL} = c^E_{KL} + \frac{[e_{Kj} l_j][l_i e_{iL}]}{l_i \epsilon^S_{ij} l_j}.$$

We can explore this for different crystal cuts with piezo effects both disabled and active.

[208]:
```
mat_LiNbO3 = materials.make_material("LiNbO3_1973_Auld")
#mat_LiNbO3 = materials.make_material("LiNbO3aniso_2021_Steel")

mat_LiNbO3_x = mat_LiNbO3.copy()
mat_LiNbO3_x.disable_piezoelectric_effects()
mat_LiNbO3_x.set_orientation('x-cut')
```

[35]:
```
print(mat_LiNbO3_x.elastic_properties())
```

```
Elastic properties of material LiNbO3_2023_Rodrigues
  Density:        4650.000 kg/m^3
  Ref. index:     2.2100+0.0000j
  Crystal class:  Trigonal
  Crystal group:  no sym
  Stiffness c_IJ:
 Stiffness c_IJ, unit: GPa.
   Voigt 4-tensor:
    [[198.8300  54.6400  68.2300  -0.0000   7.8300   0.0000]
     [ 54.6400 198.8300  68.2300   0.0000  -7.8300  -0.0000]
     [ 68.2300  68.2300 235.7100   0.0000   0.0000   0.0000]
     [ -0.0000   0.0000   0.0000  59.8600   0.0000  -7.8300]
     [  7.8300  -7.8300   0.0000   0.0000  59.8600  -0.0000]
     [  0.0000  -0.0000   0.0000  -7.8300  -0.0000  72.0950]]

  Wave mode 1: v_p=7.1197 km/s,  |v_g|=7.1197 km/s,  u_j=[ 0.0000  0.0000  1.0000], ␣
↪v_g=[ 0.0000  0.0000  7.1197] km/s
  Wave mode 2: v_p=3.5879 km/s,  |v_g|=3.6185 km/s,  u_j=[ 1.0000  0.0000  0.0000], ␣
↪v_g=[ 0.4693 -0.0000  3.5879] km/s
  Wave mode 3: v_p=3.5879 km/s,  |v_g|=3.6185 km/s,  u_j=[ 0.0000  1.0000  0.0000], ␣
↪v_g=[-0.4693  0.0000  3.5879] km/s
```

### Piezo effects disabled

Starting with piezo effects disabled, the slowness curve (top right) in the first plot below corresponds to Fig. 8.9 in Auld volume 1.

[209]:
```
imgoffx = mat_LiNbO3_x.plot_bulk_dispersion_all("tmp_xcut_piezo_off", show_poln=False,
↪ flip_x=True)
img_pair(imgoffx, "images/auld_fig8_9.png",  'X-cut, no piezo - NumBAT', 'X-cut, no␣
↪piezo - Auld Fig. 8.9', 375, 300)
```

```
<IPython.core.display.HTML object>
```

To make this comparison clearer, we can plot just the slowness curve (inverse phase velocity) in the top right. The `flip_x=True` flag reverses the sense of the $x$-axis. This compensates for NumBAT plotting dispersion in the $q_x$-$q_z$ plane, while Auld uses the $q_x$-$q_y$ plane.

[210]:
```
imgoff2 = mat_LiNbO3_x.plot_bulk_dispersion_ivp("tmp_xcut_piezo_offb", show_
↪poln=False, flip_x=True)
img_pair(imgoff2, "images/auld_fig8_9.png",  'X-cut, piezo off - NumBAT', 'X cut,␣
↪piezo off - Auld Fig. 8.9', 420, 300)
```

```
<IPython.core.display.HTML object>
```

### Piezo effects enabled

[211]:
```
mat_LiNbO3_x.enable_piezoelectric_effects()
imgonx = mat_LiNbO3_x.plot_bulk_dispersion_ivp("tmp_xcut_piezo_on", show_poln=True,␣
↪flip_x=True)
img_pair(imgonx, "images/auld_fig8_11.png",  'X-cut, piezo on - NumBAT', 'X-cut,␣
↪piezo on - Auld Fig. 8.11', 400, 300)
```

```
<IPython.core.display.HTML object>
```

The slowness curve in the second plot with piezo effects enabled corresponds to Fig. 8.11 in Auld volume 1.

[212]:
```
mat_LiNbO3_z = mat_LiNbO3.copy()
mat_LiNbO3_z.set_orientation('z-cut')
#mat_LiNbO3_z.disable_piezoelectric_effects()

imgonz = mat_LiNbO3_z.plot_bulk_dispersion_ivp("tmp_zcut_piezo_on", show_poln=True,␣
↪flip_x=True)
img_pair(imgonz, "images/auld_fig8_12.png",  'Z-cut, piezo on - NumBAT', 'Z-cut,␣
↪piezo on - Auld Fig. 8.12', 400, 300)
```

```
<IPython.core.display.HTML object>
```

[ ]:

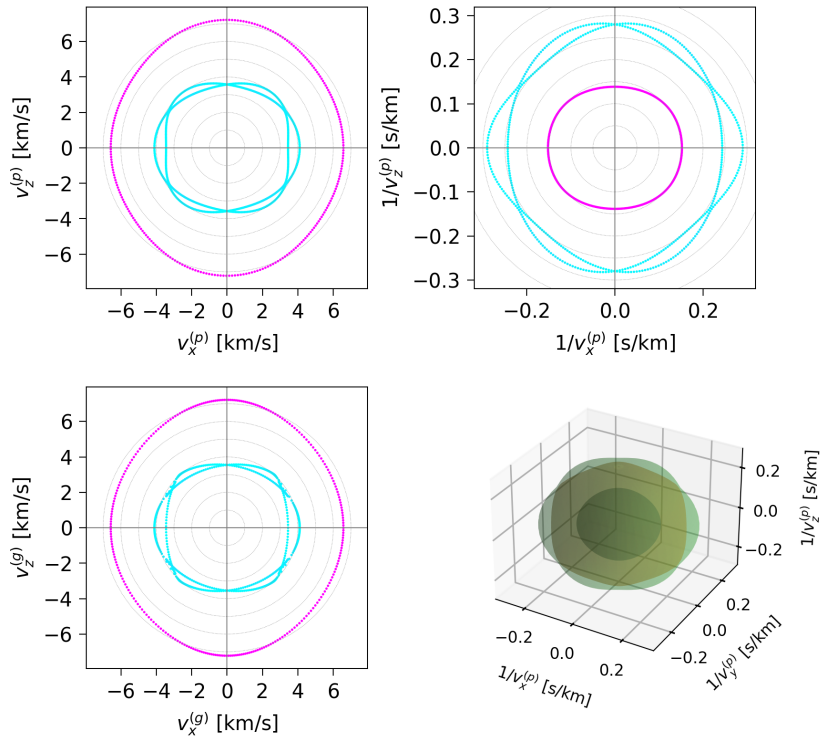
[ ]:

[ ]:

[ ]:

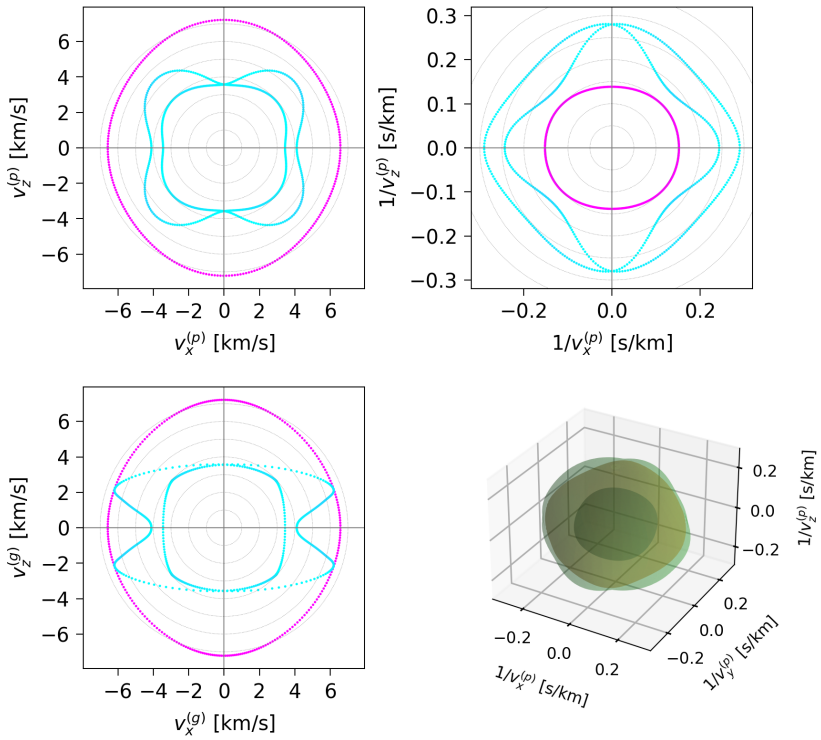
This should give Auld Fig. 8.11.

[ ]:
```
mat_LiNbO3_negx.enable_piezoelectric_effects()
imgon = mat_LiNbO3_negx.plot_bulk_dispersion("tmp_negxcut_piezo_on", show_poln=False)
display(Image(imgon, width=500))
```




This should give Auld Fig. 8.12.

[191]:
```
mat_LiNbO3_z = mat_LiNbO3.copy()
mat_LiNbO3_z.set_orientation('z-cut')

mat_LiNbO3_z.enable_piezoelectric_effects()
imgon = mat_LiNbO3_z.plot_bulk_dispersion("tmp_negxcut_piezo_on", show_poln=False)
display(Image(imgon, width=500))
```



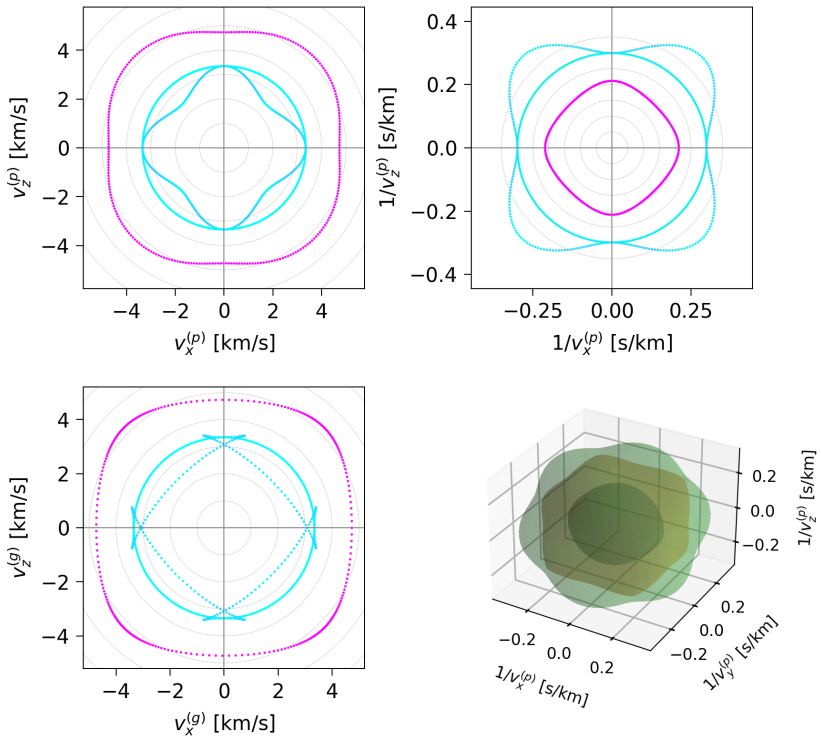

## 7.4 Other material comparisons

Cubic material: GaAs

[171]:
```
mat_GaAs = materials.make_material("GaAs_1970_Auld")
```

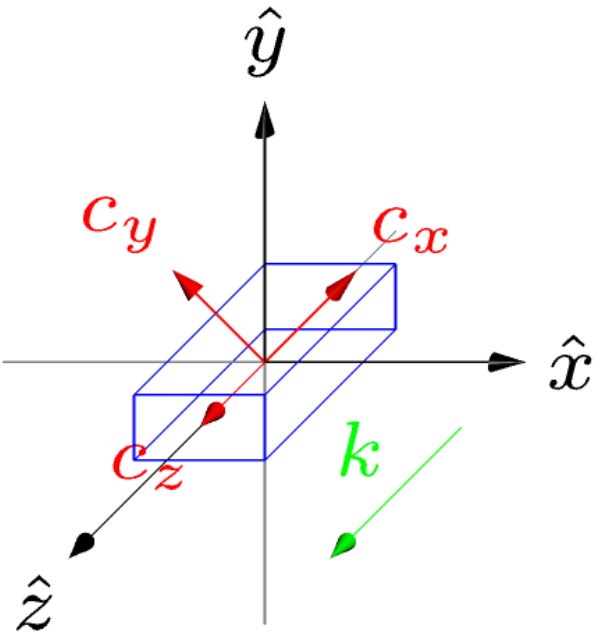
The slowness plot (top-right) below reproduces Fig. 7.2 of Auld vol. 1, while the ray surface plot (lower-left) reproduces Fig. 7.9.

[172]:
```
mat_GaAs.disable_piezoelectric_effects()

fimg = mat_GaAs.plot_bulk_dispersion("tmp_defcut", show_poln=False)
display(Image(fimg, width=500))
```

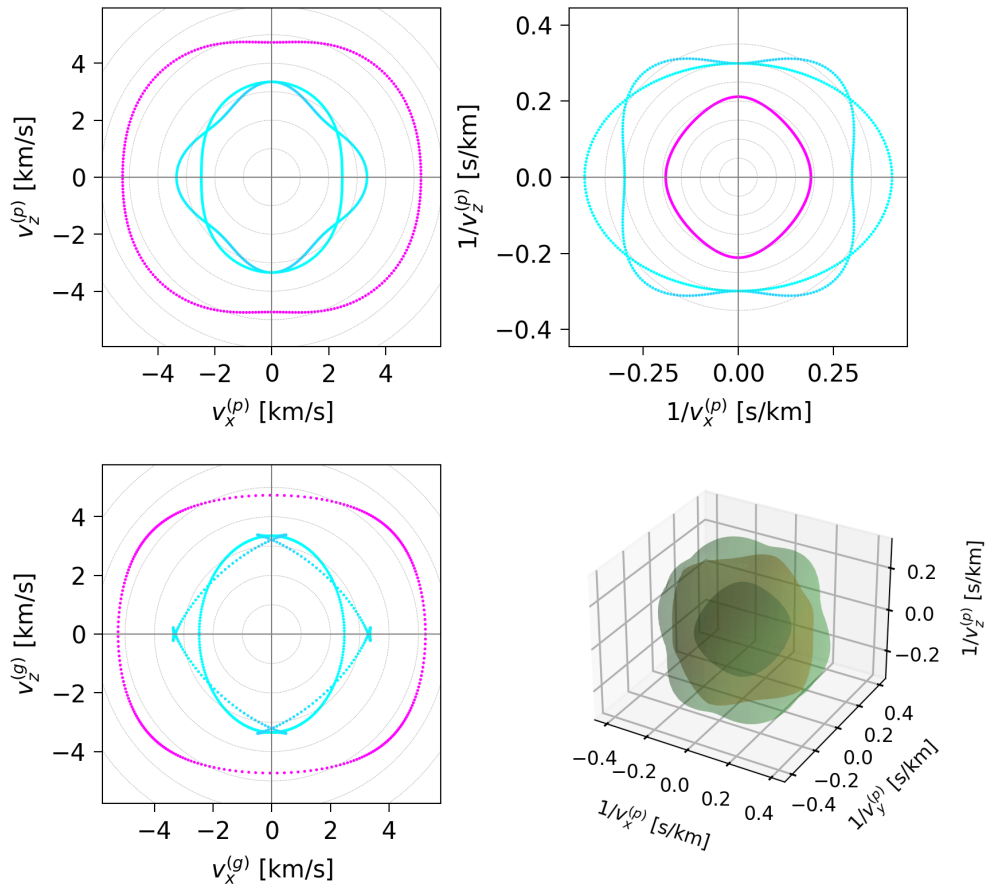



Now let's reproduce Fig. 7.3. To preserve the $z$ axis, rotate $[1, -1, 0]$ into the $x$ axis and $[1, -1, 1]$ into the horizontal plane, we need a $\pi/4$ rotation around $z$:

[173]:
```
mat_rot = mat_GaAs.copy()
mat_rot.rotate([0,0,1], np.pi/4)
fimg = mat_rot.make_crystal_axes_plot('ttx')
display(Image(fimg, width=300))
```



[174]:
```
fimg = mat_rot.plot_bulk_dispersion("tmp_defcut", show_poln=False)
display(Image(fimg, width=800))
```



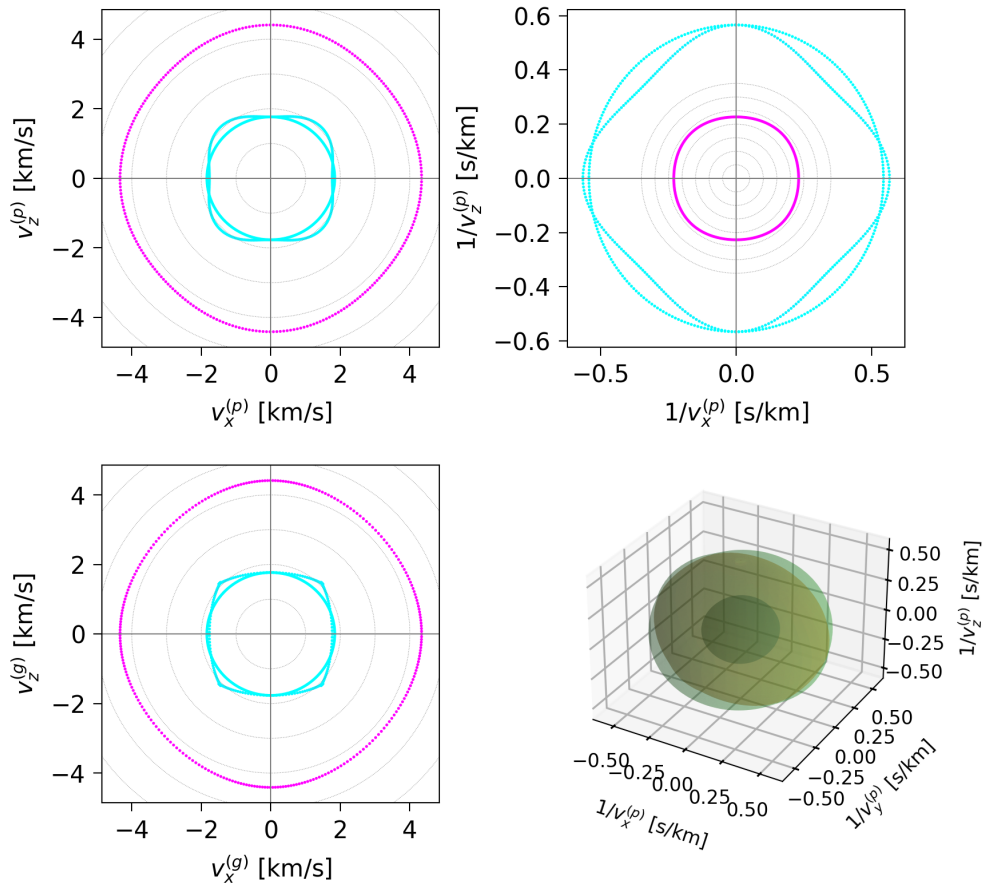

Hexagonal material: Cadmium sulfide

[185]:
```
mat_CdS = materials.make_material("CdS_1973_Auld")
mat_CdS.set_orientation('z-cut')
#print(mat_CdS.full_str())
```

We rotate to the $z$-cut orientation to get a dispersion cut through the horizontal symmetry plane.

The slowness curve (top-right) then reproduces Auld Fig. 7.6.

[187]:
```
mat_CdS.set_orientation('z-cut')
fimg = mat_CdS.plot_bulk_dispersion("tmp_defcut", show_poln=False)
display(Image(fimg, width=600))
```


$v_z^{(p)}$ [km/s]
$v_x^{(p)}$ [km/s]
$1/v_z^{(p)}$ [s/km]
$1/v_x^{(p)}$ [s/km]
$v_z^{(g)}$ [km/s]
$v_x^{(g)}$ [km/s]
$1/v_x^{(p)}$ [s/km]
$1/v_y^{(p)}$ [s/km]
$1/v_z^{(p)}$ [s/km]

[ ]:

CHAPTER

# EIGHT

# ADVANCED TUTORIALS

This second set of examples/tutorials begin to explore some more advanced features including anisotropy, complex waveguide shapes and validation of NumBAT results using analytically solvable problems.

## 8.1 Tutorial 11 – Two-layered 'Onion'

This tutorial, contained in `sim-tut_11a-onion2.py` demonstrates use of a two-layer `onion` structure for a backward intra-modal SBS configuration. Note that with the inclusion of the background layer, the two-layer onion effectively creates a three-layer geometry with core, cladding, and background surroundings. This is the ideal structure for investigating the cladding modes of an optical fibre. It can be seen by looking through the optical mode profiles in `tut_11a-fields/EM*.png` that this particular structure supports five cladding modes in addition to the three guided modes (the $TM_0$ mode is very close to cutoff).

Next, the gain spectrum and the mode profiles of the main peaks indicate as expected, that the gain is optimal for modes that are predominantly longitudinal in character.

The accompanying tutorial `sim-tut_11b-onion3.py` introduces one additional layer and would be suitable for studying the influence of the outer polymer coating of an optical fibre or depressed cladding W fibre.

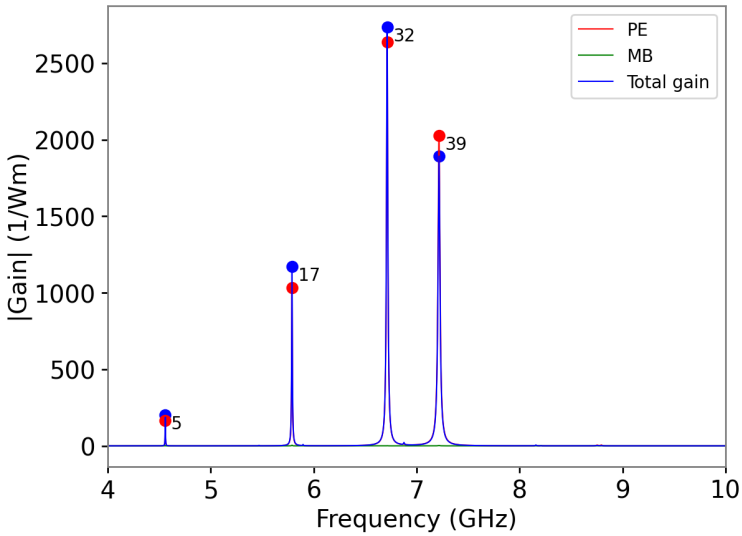



Fig. 1: Gain spectrum for the two-layer structure in `tut_11a`.

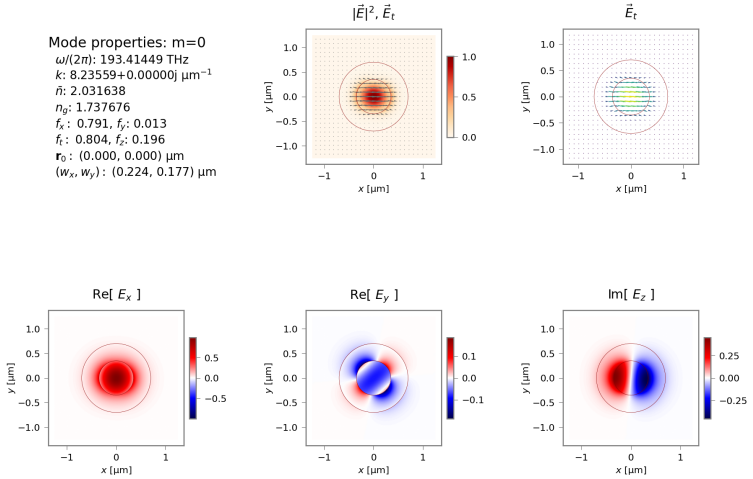



Fig. 2: Mode profile for fundamental optical mode.

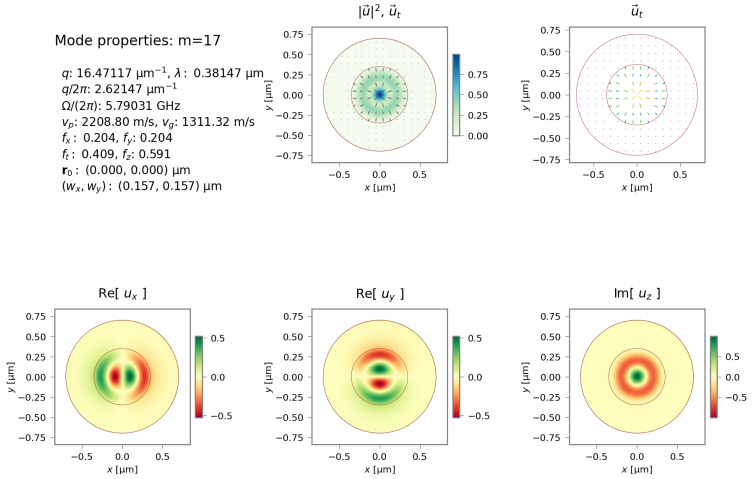



Fig. 3: Mode profile for acoustic mode 17.

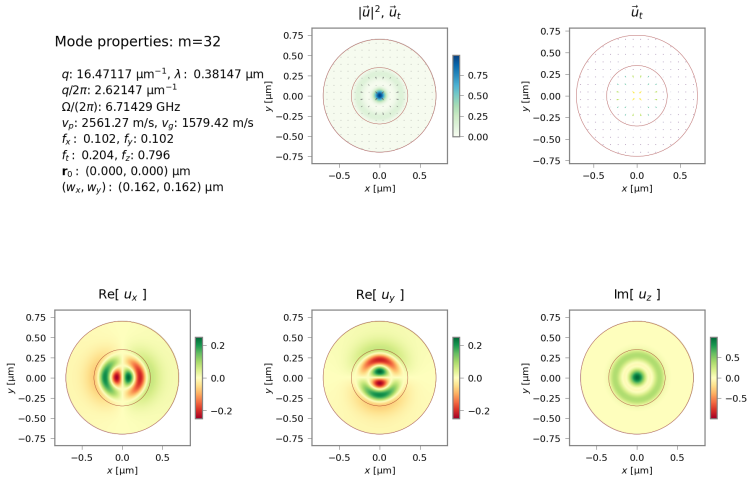



Fig. 4: Mode profile for acoustic mode 32.

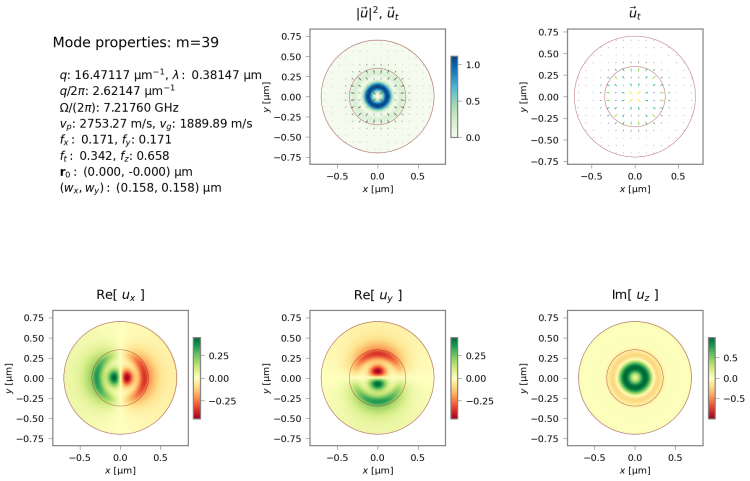



Fig. 5: Mode profile for acoustic mode 39.

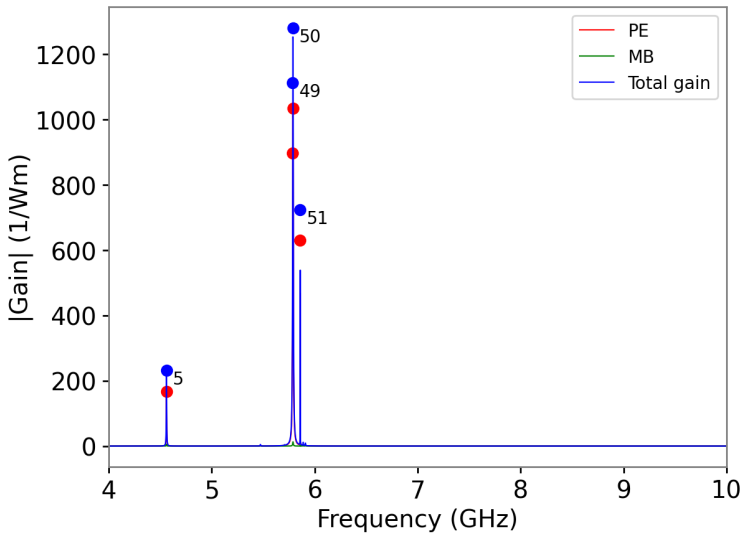



Fig. 6: Gain spectrum for the three-layer structure in `tut_11b`.

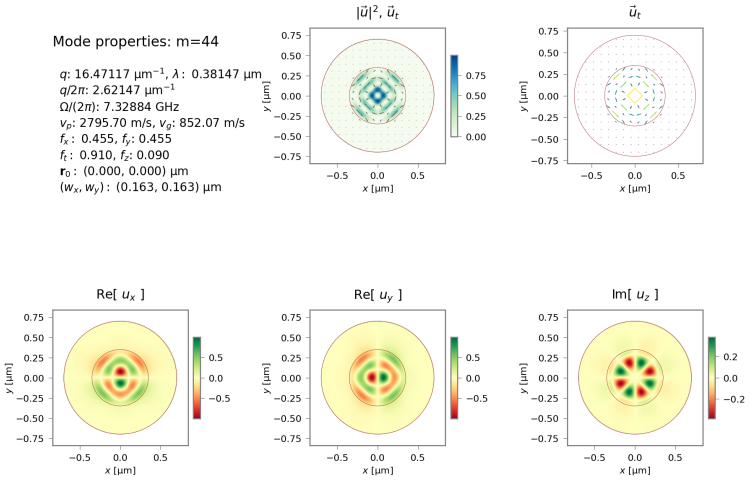



Fig. 7: Mode profile for acoustic mode 44.

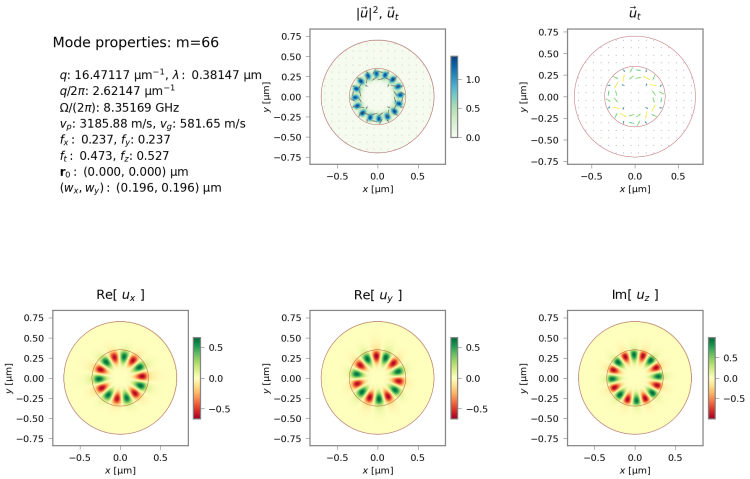



Fig. 8: Mode profile for acoustic mode 66.

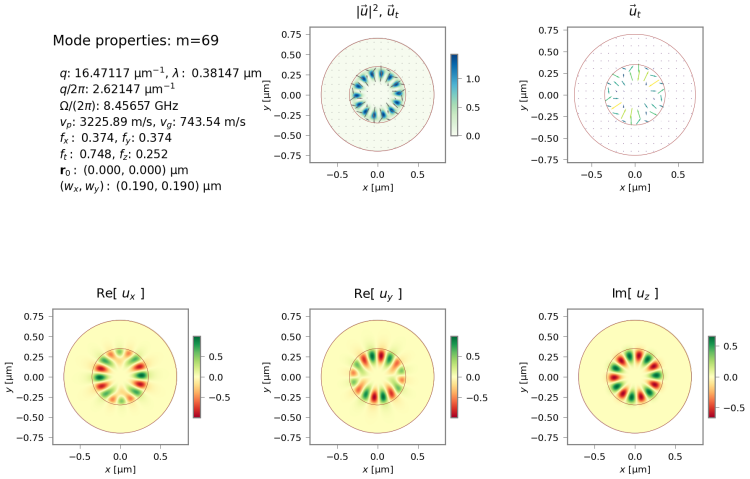



Fig. 9: Mode profile for acoustic mode 66.

## 8.2 Tutorial 12 – Validating the calculation of the EM dispersion of a two-layer fibre

How can we be confident that NumBAT's calculations are actually correct? This tutorial and the next one look at rigorously validating some of the modal calculations produced by NumBAT.

This tutorial, contained in `sim-tut_12.py`, compares analytic and numerical calculations for the dispersion relation of the electromagnetic modes of a cylindrical waveguide. This can be done in both a low-index contrast (SMF-28 fibre) and high-index contrast (silicon rod in silica) context. We calculate the effective index $\bar{n}$ and normalised waveguide parameter $b = (\bar{n}^2 - n_{\mathrm{cl}}^2)/(n_{\mathrm{co}}^2 - n_{\mathrm{cl}}^2)$ as a function of the normalised freqency $V = ka\sqrt{n_{\mathrm{co}}^2 - n_{\mathrm{cl}}^2}$ for radius $a$ and wavenumber $k = 2\pi/\lambda$. As in several previous examples, this is accomplished by a scan over the wavenumber $k$.

The numerical results (marked with crosses) are compared to the modes found from the roots of the rigorous analytic dispersion relation (solid lines). We also show the predictions for the group index $n_g = \bar{n} + V\frac{d\bar{n}}{dV}$. The only noticeable departures are right at the low $V$-number regime where the fields become very extended into the cladding and interact significantly with the boundary. The results could be improved in this regime by choosing a larger domain at the expense of a longer calculation.

As this example involves the same calculation at many values of the wavenumber $k$, we again use parallel processing techniques. However, in this case we demonstrate the use of *threads* (multiple simultaneous strands of execution within the same process) rather than a pool of separate processes. Threads are light-weight and can be started more efficiently than separate processes. However, as all threads share the same memory space, some care is needed to prevent two threads reading or writing to the same data structure simultaneously. This is dealt with using the helper functions and class in the *numbattools.py* module.

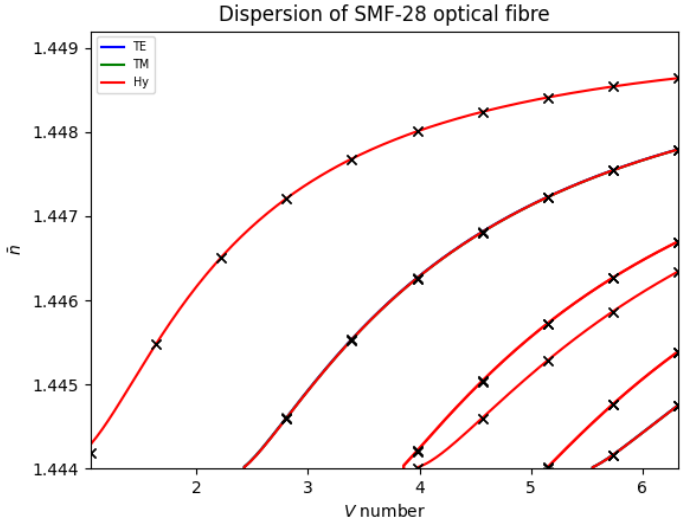



Fig. 10: Optical effective index as a function of normalised frequency for SMF-28 fibre.

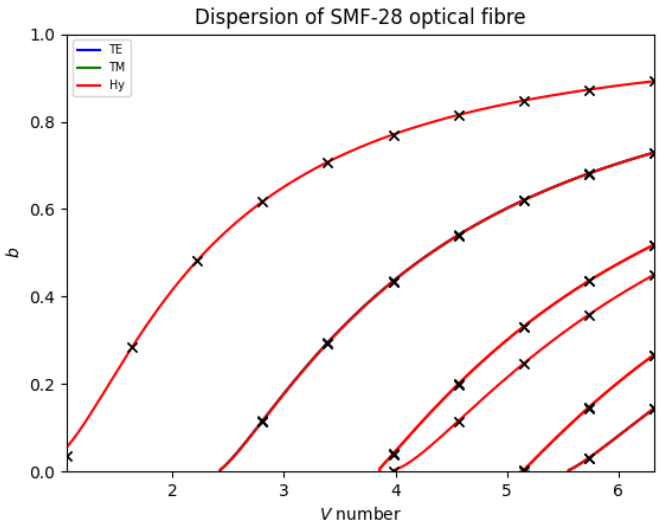



Fig. 11: Optical normalised waveguide parameter as a function of normalised frequency for SMF-28 fibre.

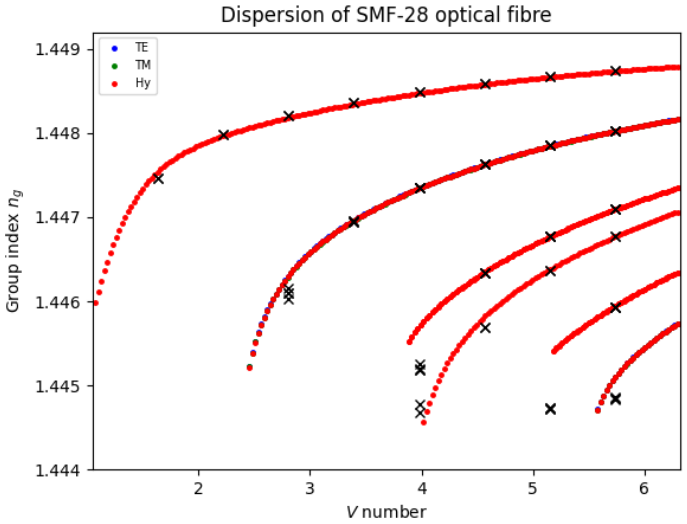



Fig. 12: Optical group index $n_g = \bar{n} + V\frac{d\bar{n}}{dV}$ as a function of normalised frequency for SMF-28 fibre.

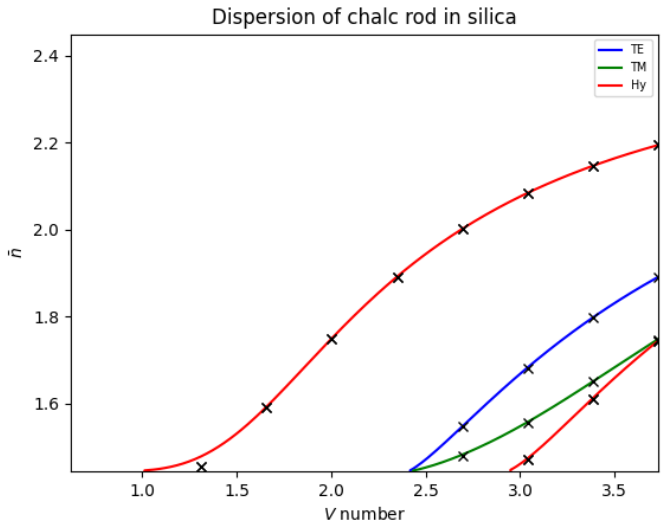



Fig. 13: Optical effective index as a function of normalised frequency for silicon rod in silica.

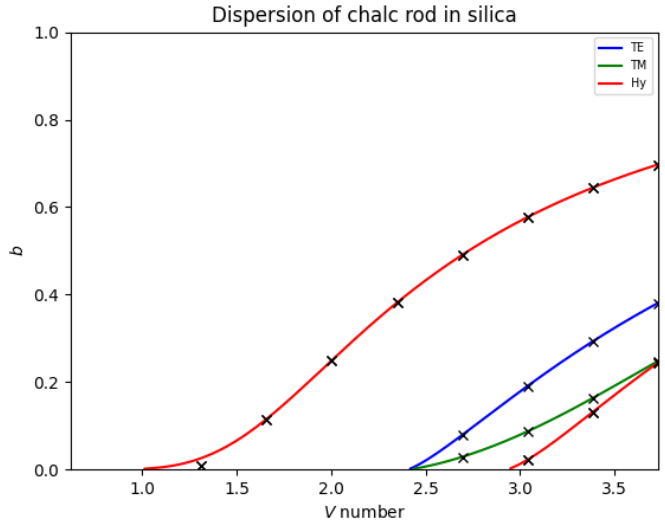



Fig. 14: Optical normalised waveguide parameter as a function of normalised frequency for silicon rod in silica.

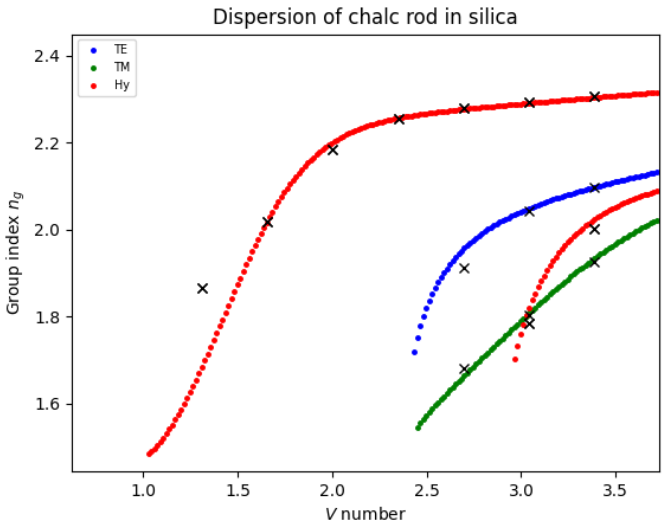



Fig. 15: Optical group index $n_g = \bar{n} + V \frac{d\bar{n}}{dV}$ as a function of normalised frequency for silicon rod in silica.

## 8.3 Tutorial 13 – Validating the calculation of the dispersion of an elastic rod in vacuum

The tutorial `sim-tut_13.py` performs the same kind of calculation as in the previous tutorial for the acoustic problem. In this case there is no simple analytic solution possible for the two-layer cylinder. Instead we create a structure of a single elastic rod surrounded by vacuum. NumBAT removes the vacuum region and imposes a free boundary condition at the boundary of the rod. The modes found are then compared to the analytic solution of a free homogeneous cylindrical rod in vacuum.

We find excellent agreement between the analytical (coloured lines) and numerical (crosses) results. Observe the existence of two classes of modes with azimuthal index $p = 0$, corresponding to the pure torsional modes, which for the lowest band propagate at the bulk shear velocity, and the so-called Pochammer hybrid modes, which are predominantly longitudinal, but must necessarily involve some shear motion to satisfy mass conservation.

It is instructive to examine the mode profiles in `tut_13-fields` and track the different field profiles and degeneracies found for each value of $p$. By basic group theory arguments, we know that every mode with $p \neq 0$ must come as a degenerate pair and this is satisfied to around 5 significant figures in the calculated results. It is interesting to repeat the calculation with a silicon (cubic symmetry) rod rather than chalcogenide (isotropic). In that case, the lower symmetry of silicon causes splitting of a number of modes, so that a larger number of modes are found to be singly degenerate, though invariably with a partner state at a nearby frequency.

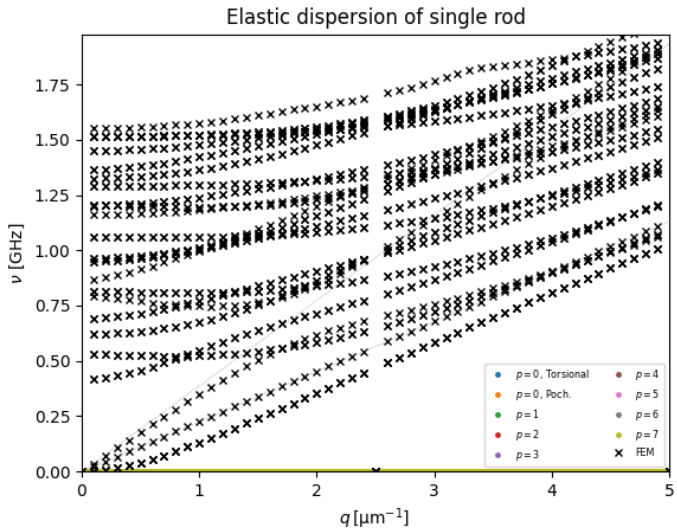



Fig. 16: Elastic frequency as a function of normalised wavenumber for a chalcogenide rod in vacuum.

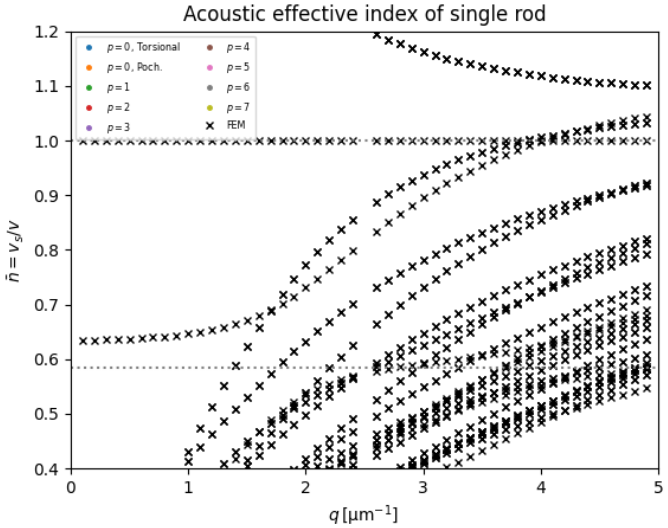



Fig. 17: Elastic "effective index" defined as the ratio of the bulk shear velocity to the phase velocity $n_{\text{eff}} = V_s/V$, for a chalcogenide rod in vacuum.

## 8.4 Tutorial 14 – Multilayered 'Onion'

** This tutorial is under development. Expect it to fail.**

This tutorial, contained in `sim-tut_14-multilayer-fibre.py` shows how one can create a circular waveguide with many concentric layers of different thickness. In this case, the layers are chosen to create a concentric Bragg grating of alternating high and low index layers. As shown in C. M. de Sterke, I. M. Bassett and A. G. Street, "Differential losses in Bragg fibers", J. Appl. Phys. **76**, 680 (1994), the fundamental mode of such a fibre is the fully azimuthally symmetric $\mathrm{TE}_0$ mode rather than the usual $\mathrm{HE}_{11}$ quasi-linearly polarised mode that is found in standard two-layer fibres.

## 8.5 Tutorial 15 – Coupled waveguides

This tutorial, contained in `examples/tutorials/sim-tut_14-coupled-wg.py` demonstrates the supermode behaviour of both electromagnetic and elastic modes in a pair of closely adjacent waveguides.

CHAPTER

NINE

# JOSA-B TUTORIAL PAPER

## 9.1 Introduction

Dr Christian Wolff and colleagues have used NumBAT throughout their 2021 SBS tutorial paper Brillouin scattering—theory and experiment: tutorial published in J. Opt. Soc. Am. B. This set of examples works through their discussions of backward SBS, forward Brillouin scattering, and intermodal forward Brillouin scattering.

As the calculations in this paper used NumBAT with essentially the same code in these examples/tutorials, we do not bother to include the original figures.

### 9.1.1 Example 1 – Backward SBS in a circular silica waveguide

Figure 15 in the paper shows the fundamental optical field of a silica 1 micron waveguide, with the gain and other parameters shown in Table 2. The corresponding results generated with `sim-josab-01.py` are as follows:

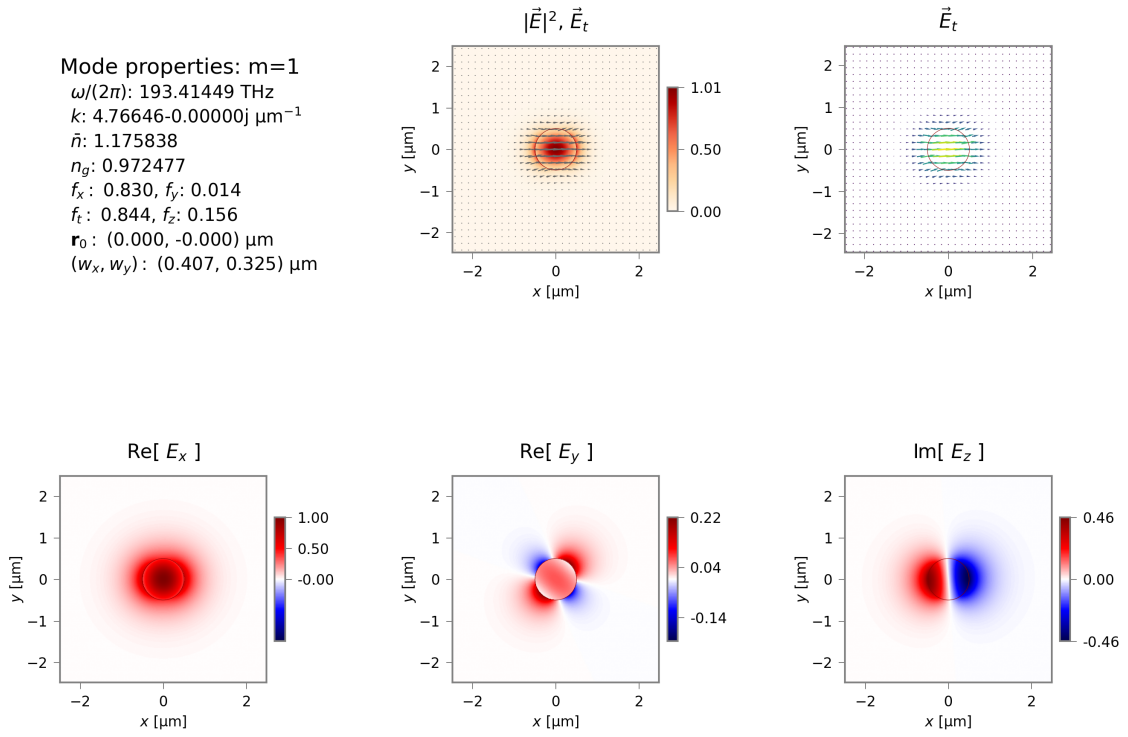



Fig. 1: Fundamental optical mode fields.

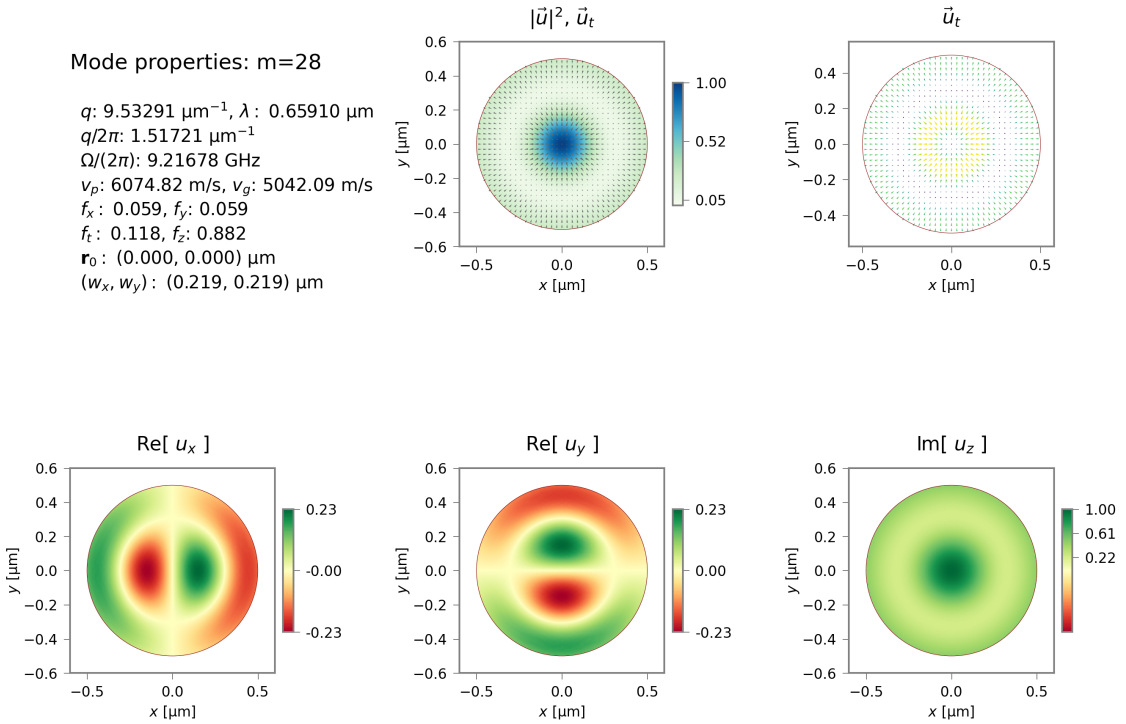



Fig. 2: Elastic mode with largest SBS gain.

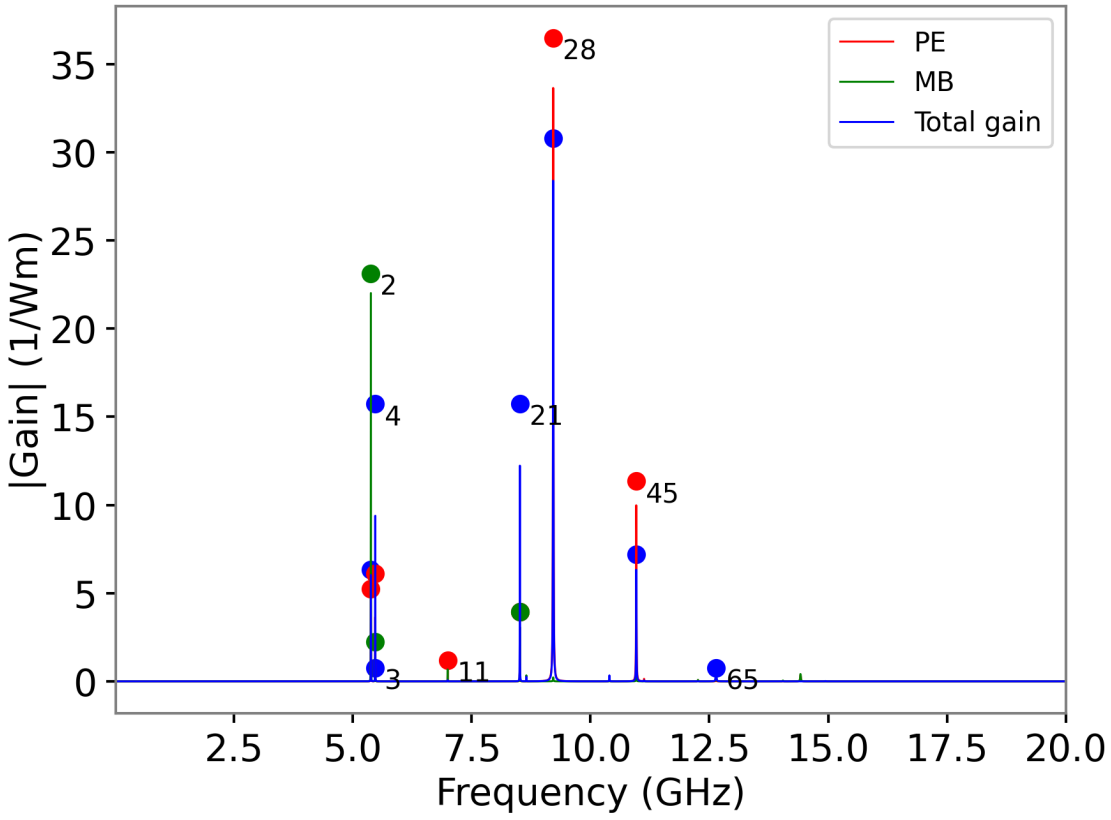
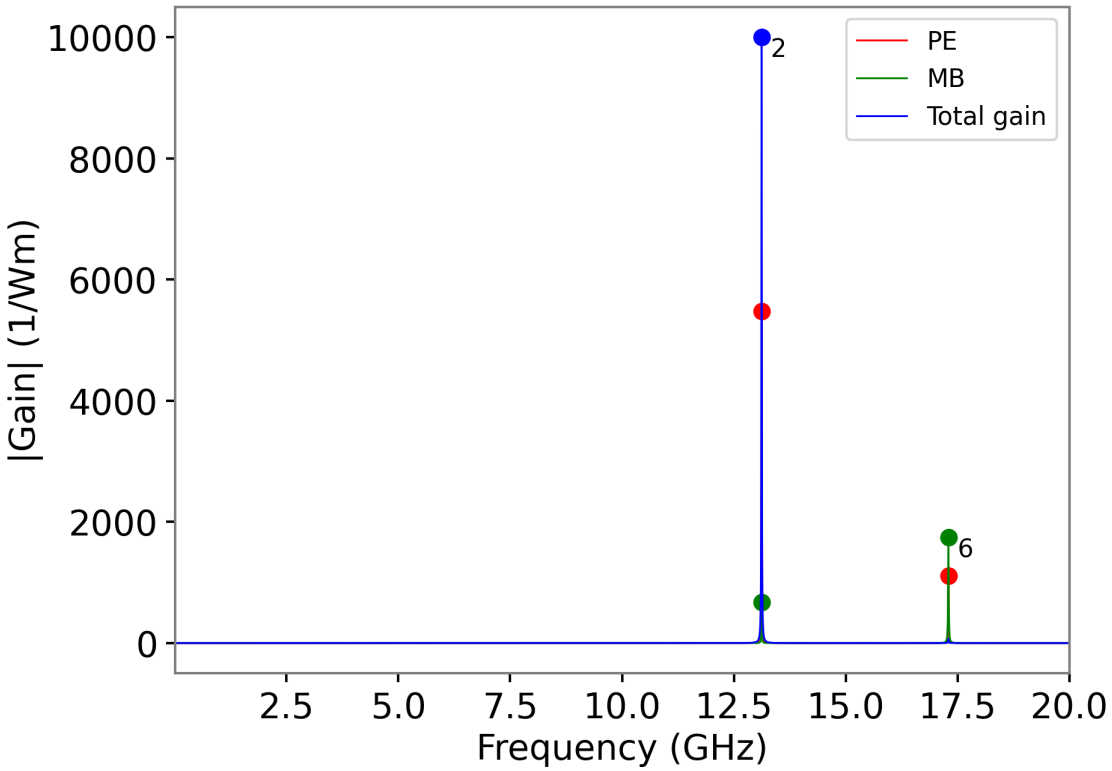



Fig. 3: Gain spectrum for the silica waveguide.

### 9.1.2 Example 2 – Backward SBS in a rectangular silicon waveguide

Figure 14 in the paper calculates the backwards SBS properties of a rectangular $450x200$ nm silicon waveguide. The corresponding results generated with `sim-josab-02.py` are as follows:




The fields and gain parameters are as follows:

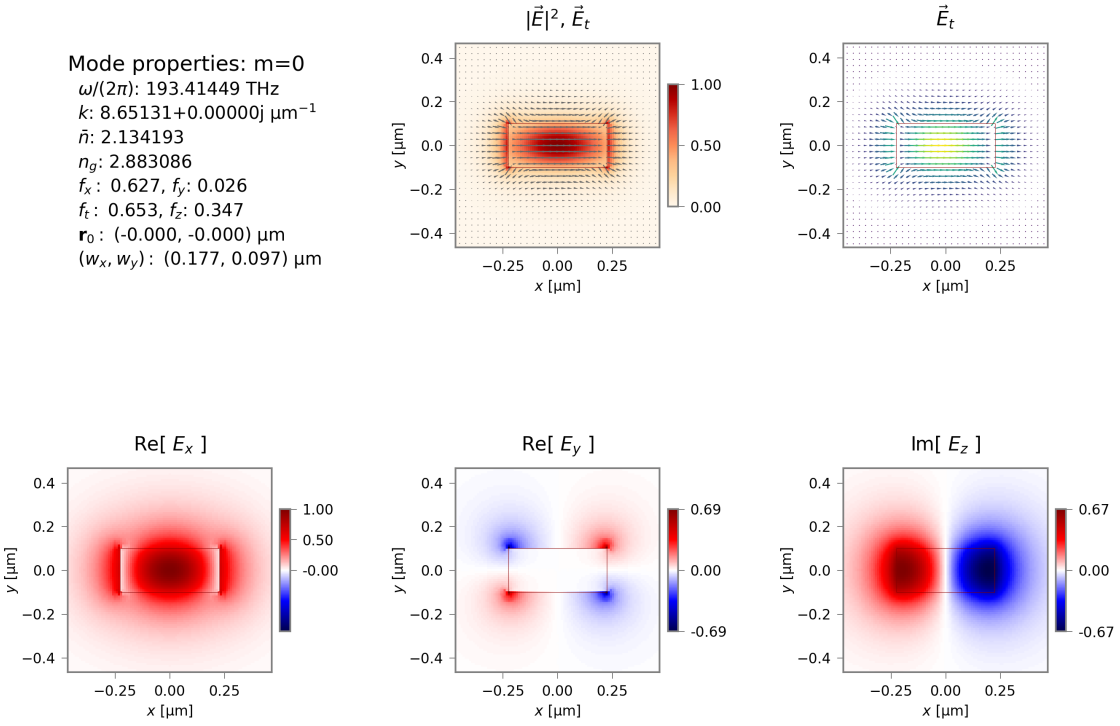



Fig. 4: Fundamental optical mode fields.

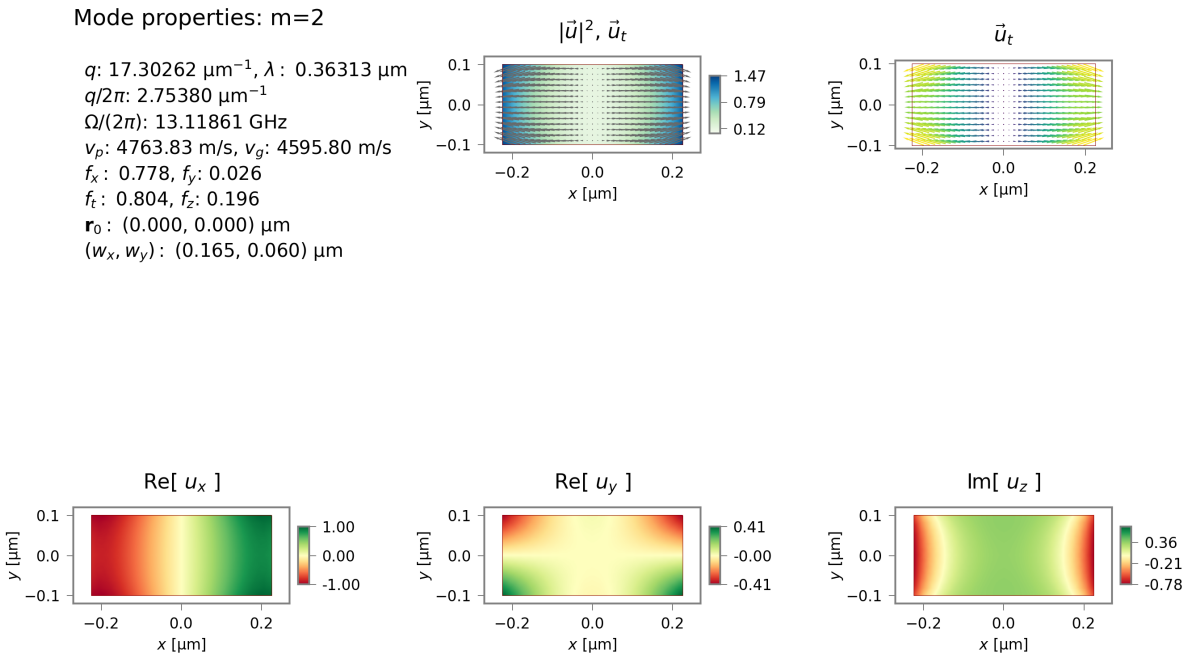



Fig. 5: Fundamental elastic mode fields for mode 2.

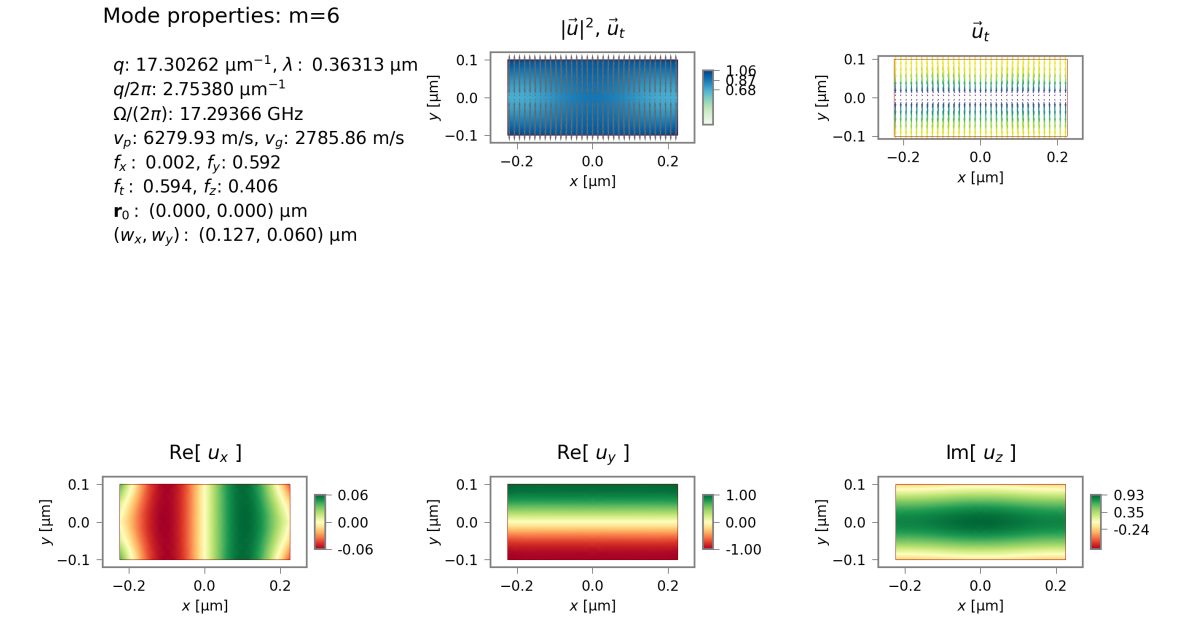



Fig. 6: Fundamental elastic mode fields for mode 6.

We can reproduce Fig. 13 showing the elastic dispersion of this waveguide silicon waveguide using `sim-josab-02b-acdisp.py`.

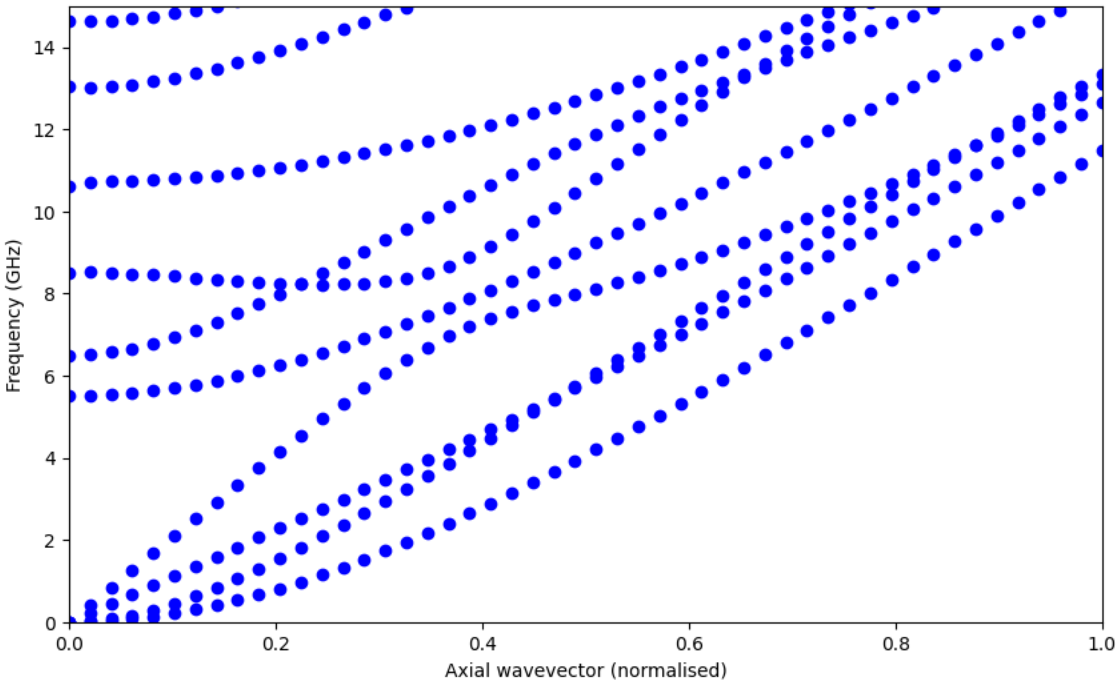



Fig. 7: Acoustic dispersion diagram with modes categorised by symmetry as in Table 1 of "Formal selection rules for Brillouin scattering in integrated waveguides and structured fibers" by C. Wolff, M. J. Steel, and C. G. Poulton `https://doi.org/10.1364/OE.22.032489`

### 9.1.3 Example 3 – Forward Brillouin scattering in a circular silica waveguide

Figure 16 and Table 3 examine the same waveguides in the case of forward Brillouin scattering.

These results can be generated with `sim-josab-03.py` and `sim-josab-04.py`.

Let's see the results for the silica cylinder first:

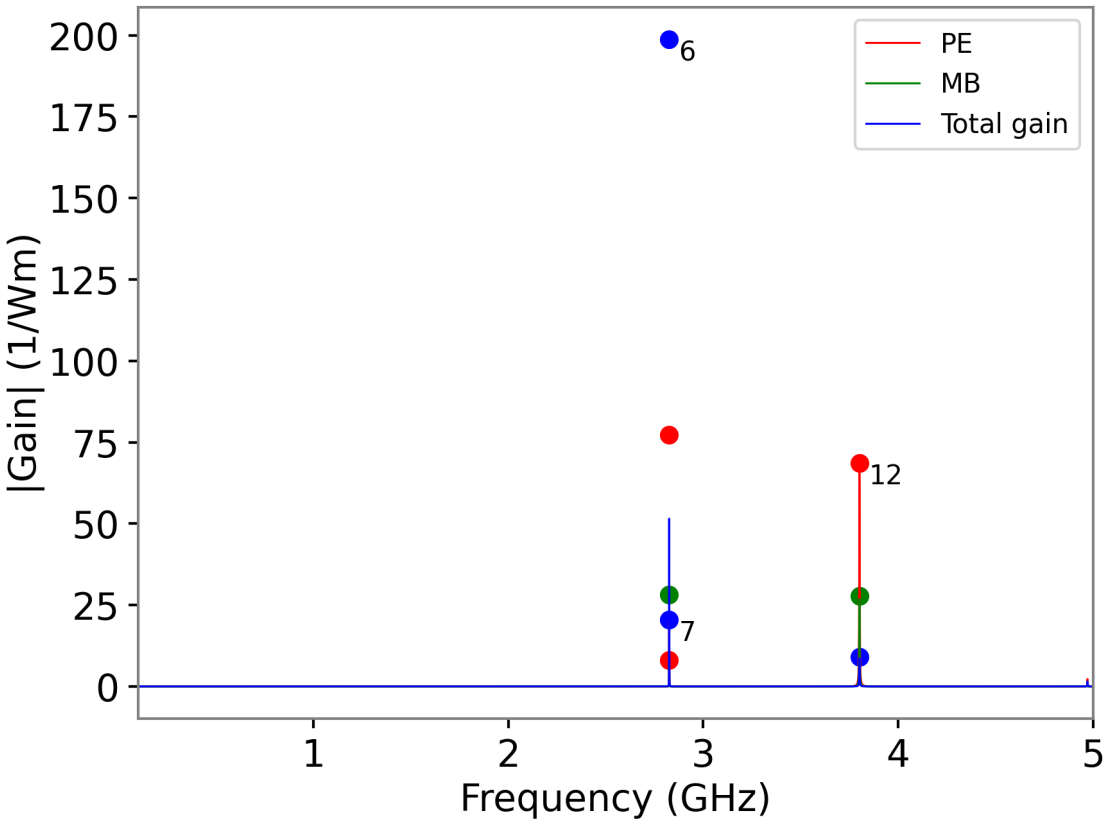



Fig. 8: Gain spectrum for forward SBS of the silica cylinder.

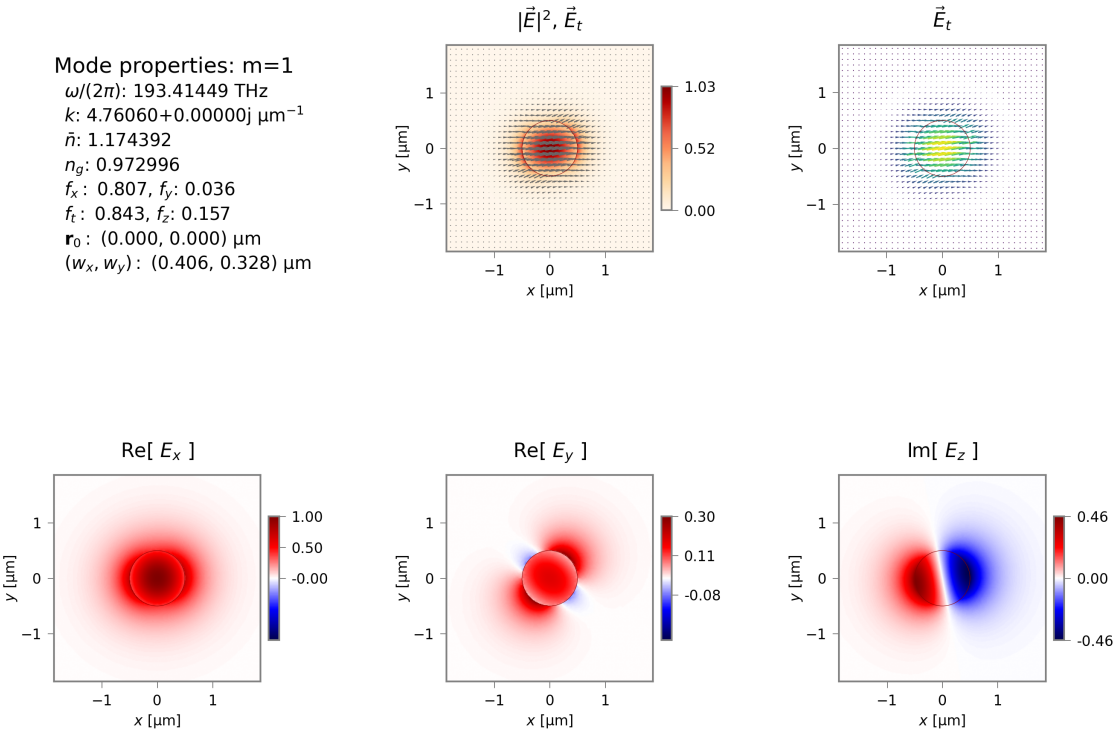



Fig. 9: Fundamental optical mode field.

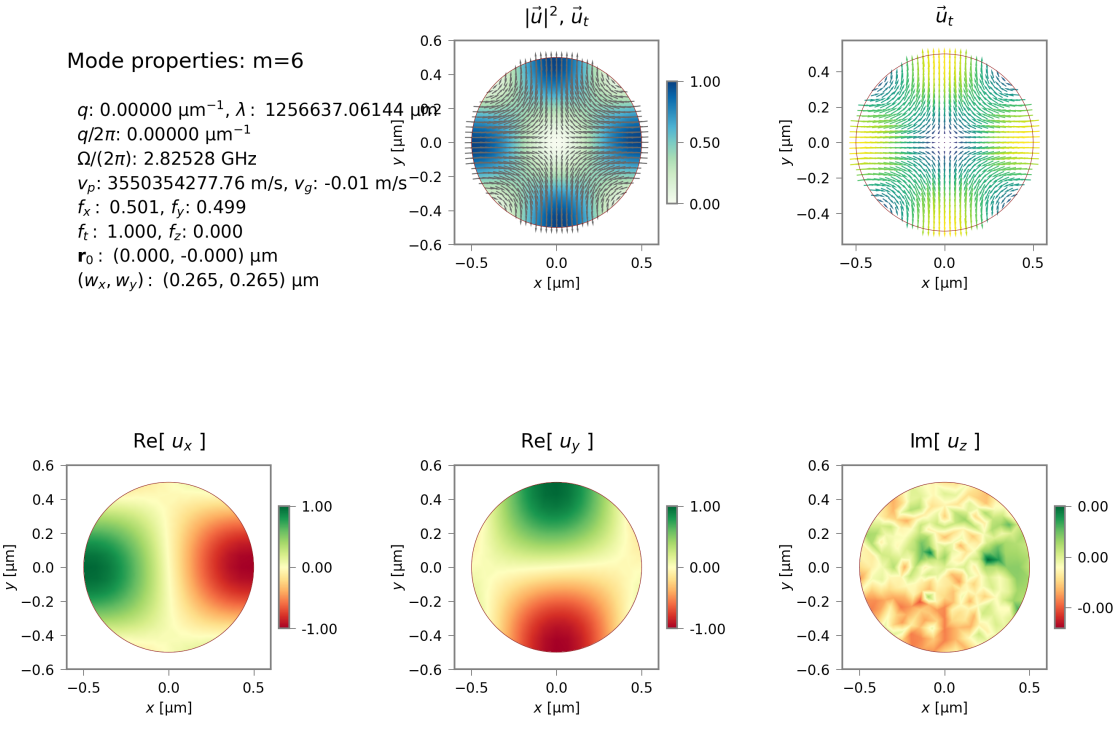



Fig. 10: Elastic mode of maximum gain.

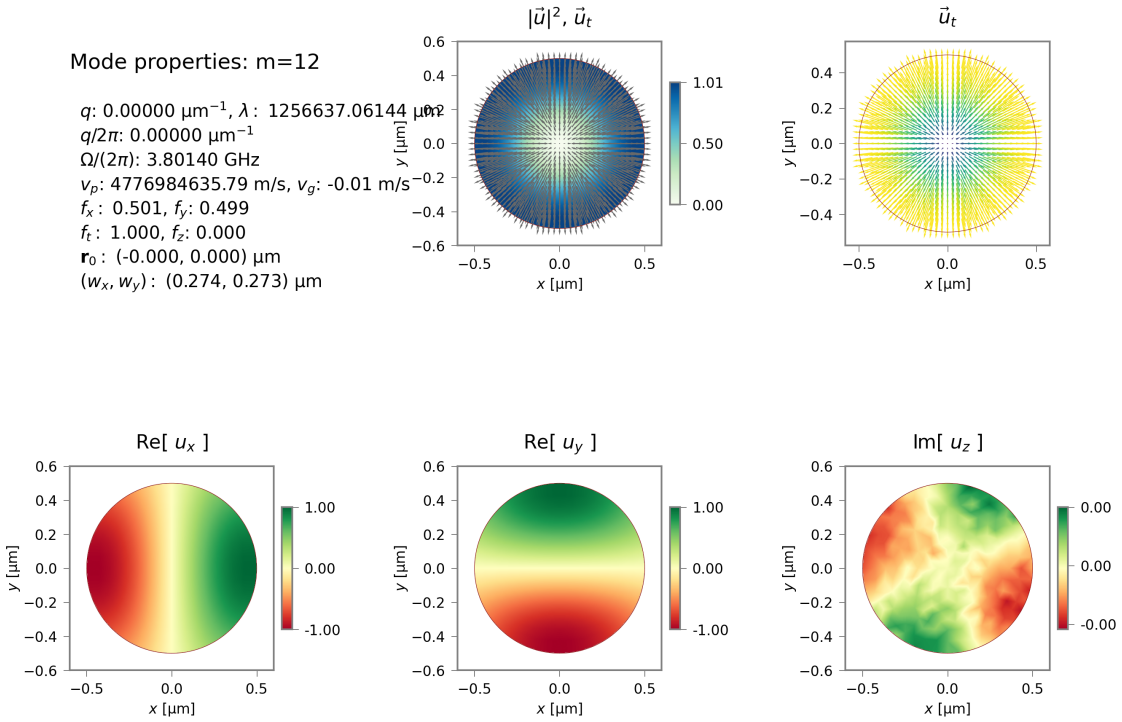



Fig. 11: Elastic mode of second highest gain.

### 9.1.4 Example 4 – Forward Brillouin scattering in a rectangular silicon waveguide

The corresponding results for the silicon waveguide can be generated with `sim-josab-04.py`:

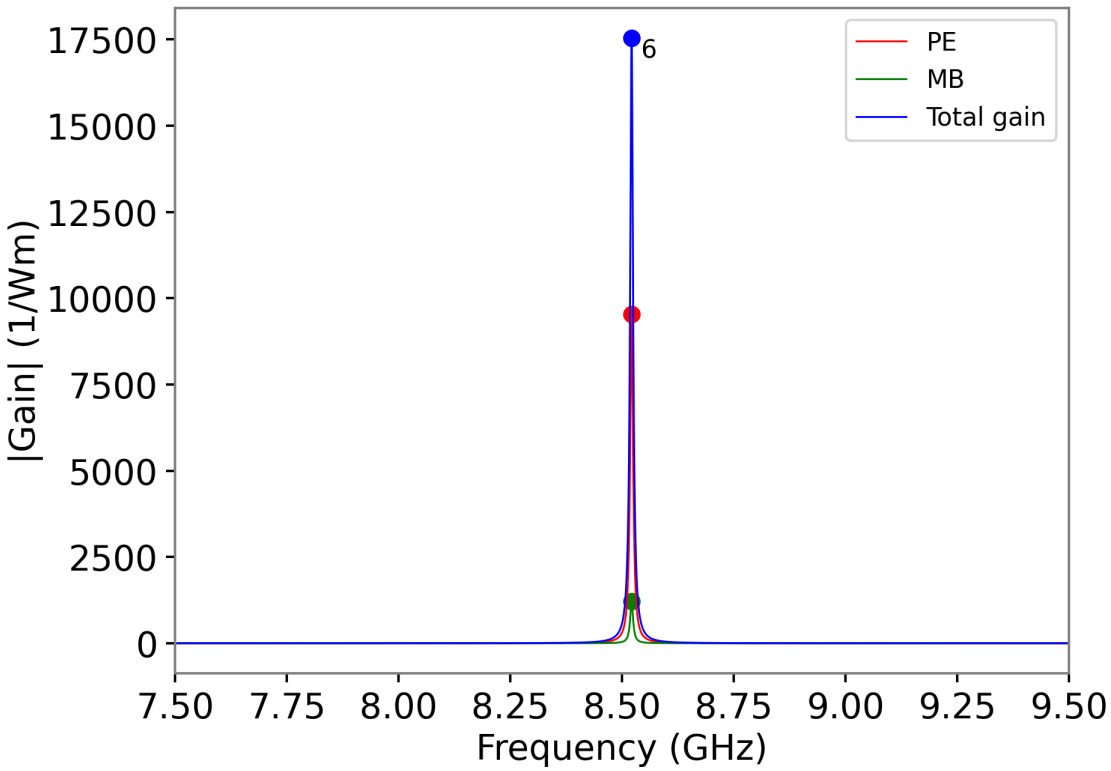



Fig. 12: Gain spectrum for forward SBS of the silicon waveguide.

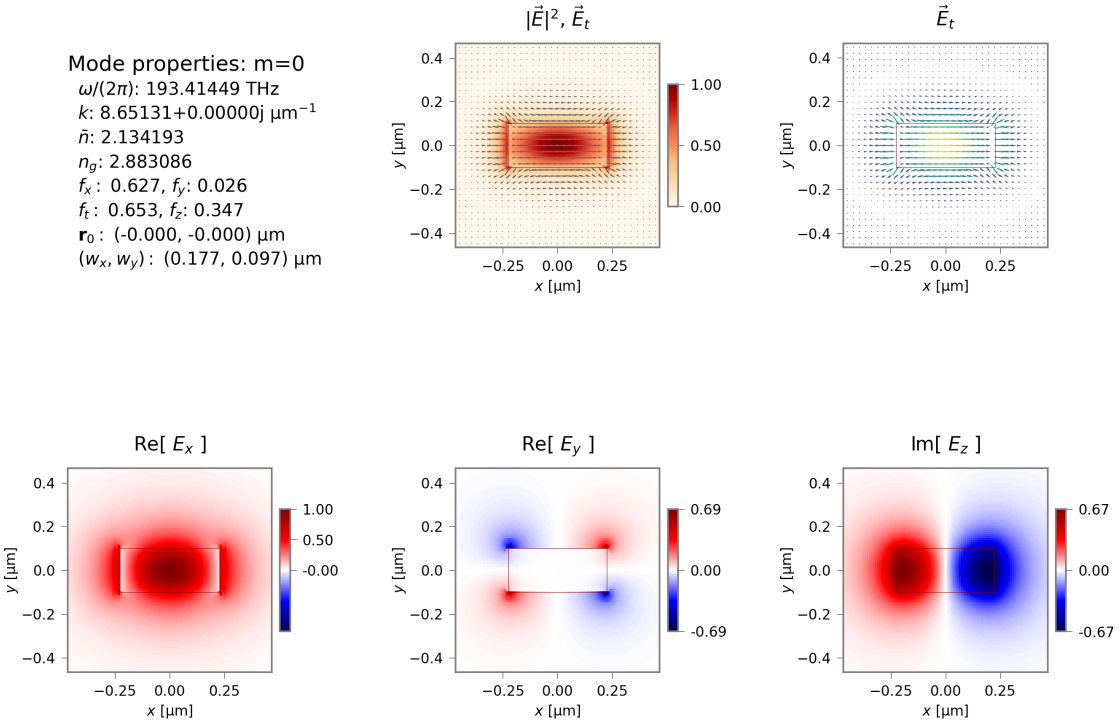



Fig. 13: Fundamental optical mode field.

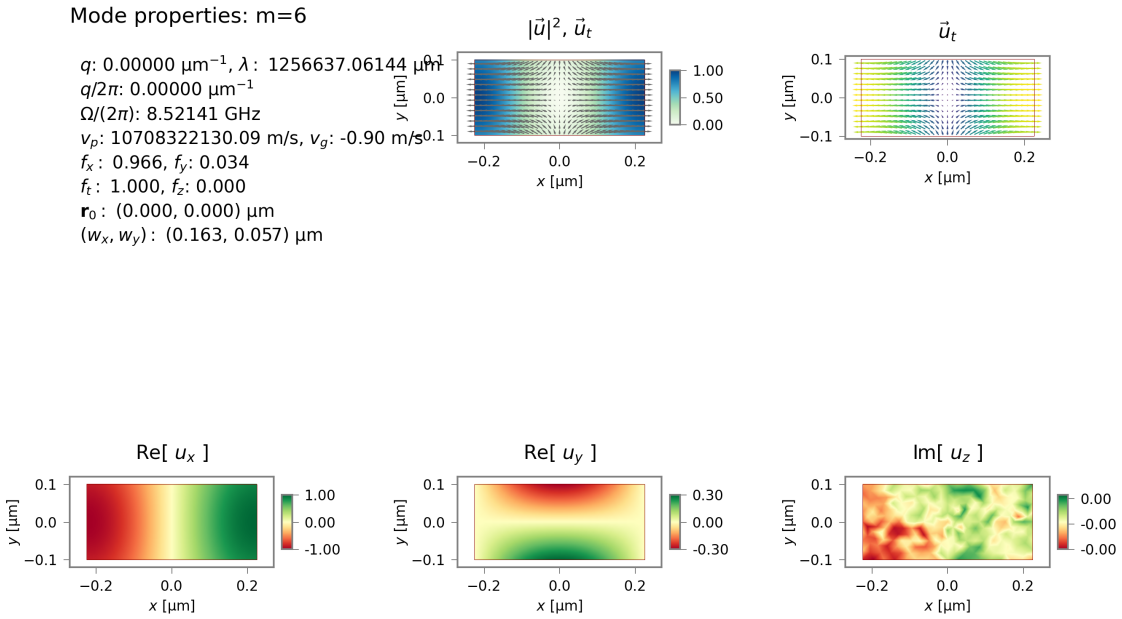



Fig. 14: Elastic mode of maximum gain.

### 9.1.5 Example 5 – Intermodal Forward Brillouin scattering in a circular silica waveguide

For the problem of intermodal FBS, the paper considers coupling between the two lowest optical modes. The elastic mode of highest gain is actually a degenerate pair:

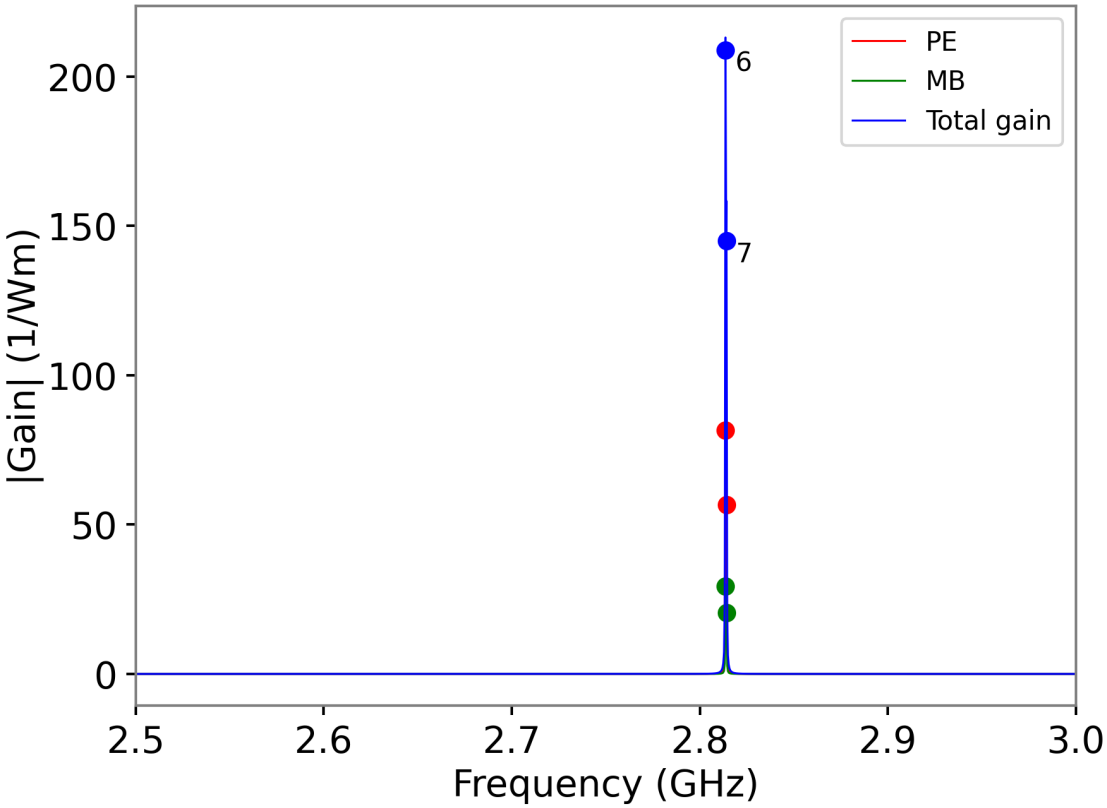



Fig. 15: Gain spectrum for intermodal forward SBS of the silica waveguide.

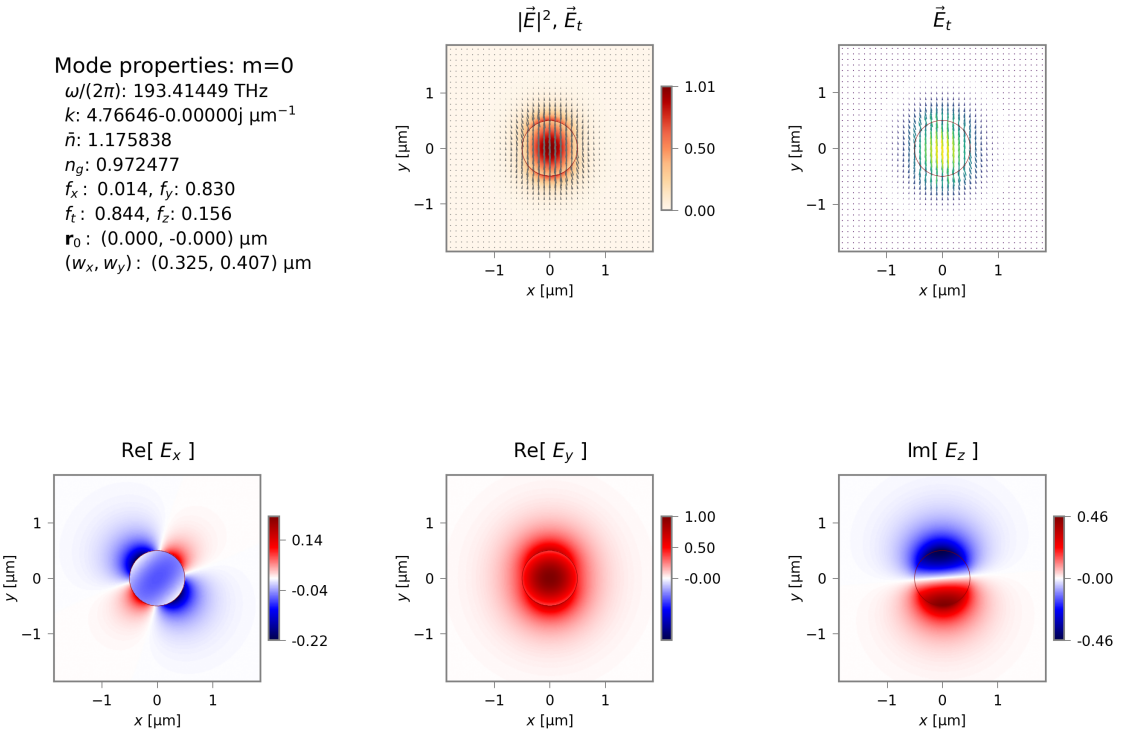



Fig. 16: Fundamental optical mode field.

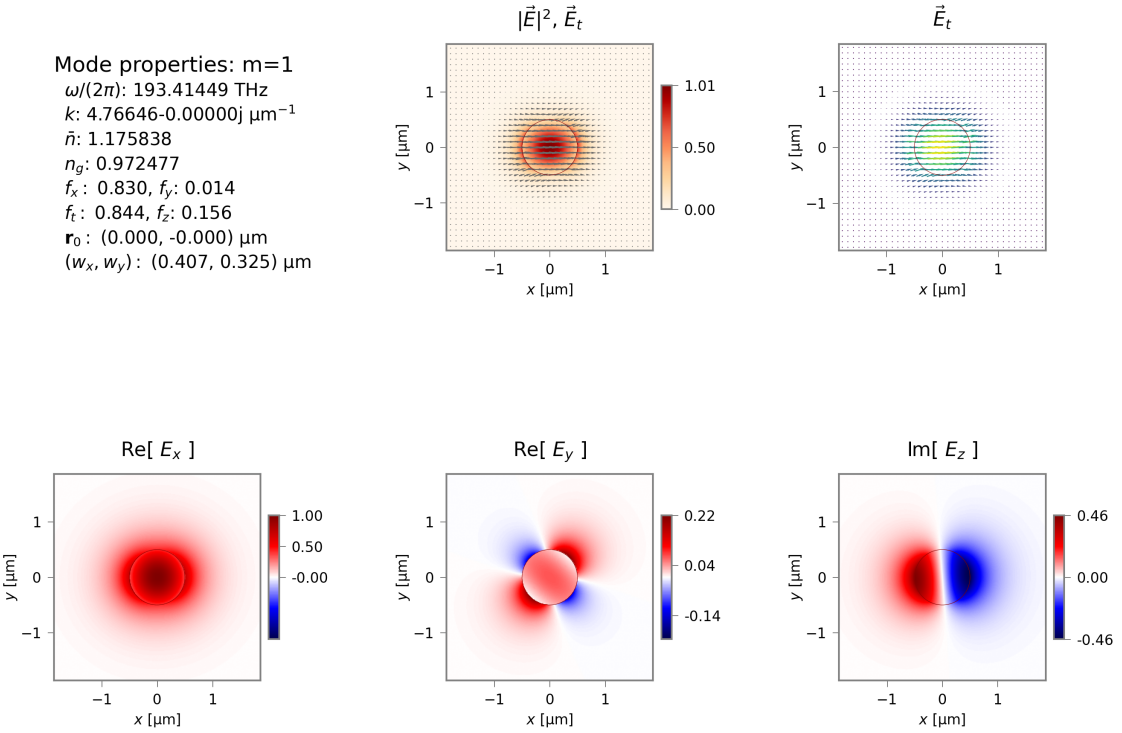



Fig. 17: Second order optical mode field.

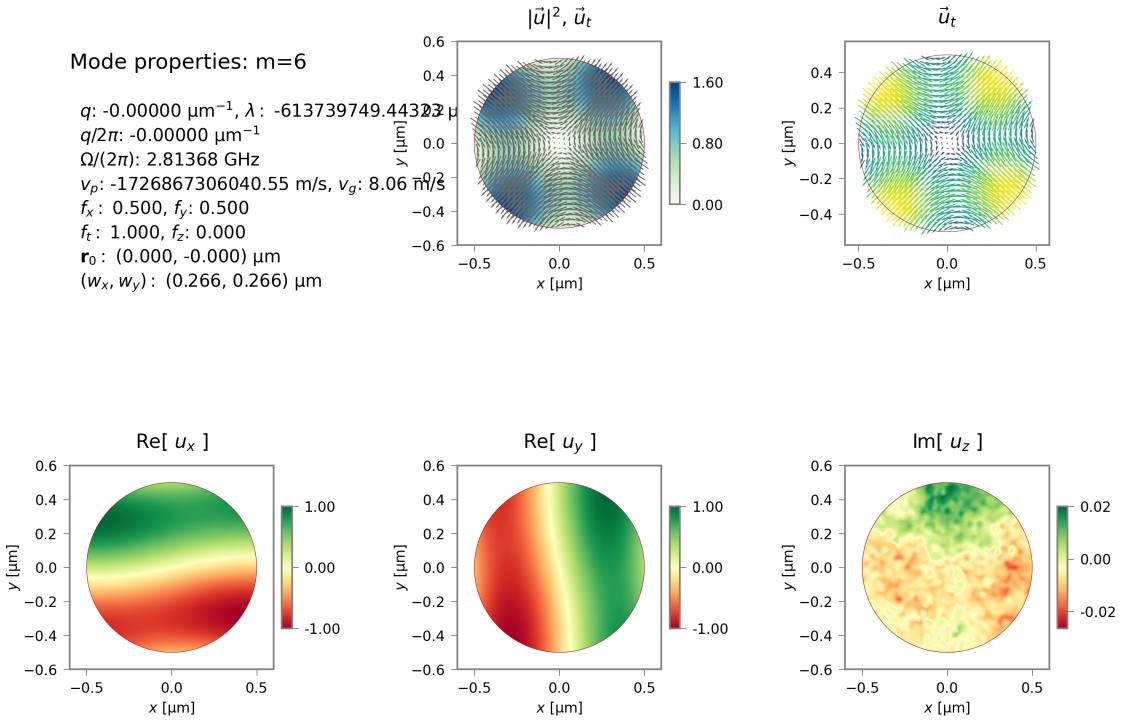



Fig. 18: Elastic mode field of maximum gain.

### 9.1.6 Example 6 – Intermodal Forward Brillouin scattering in a rectangular silicon waveguide

Finally, the silicon waveguide generates extraordinarily high gain when operated in an intermodal configuration:

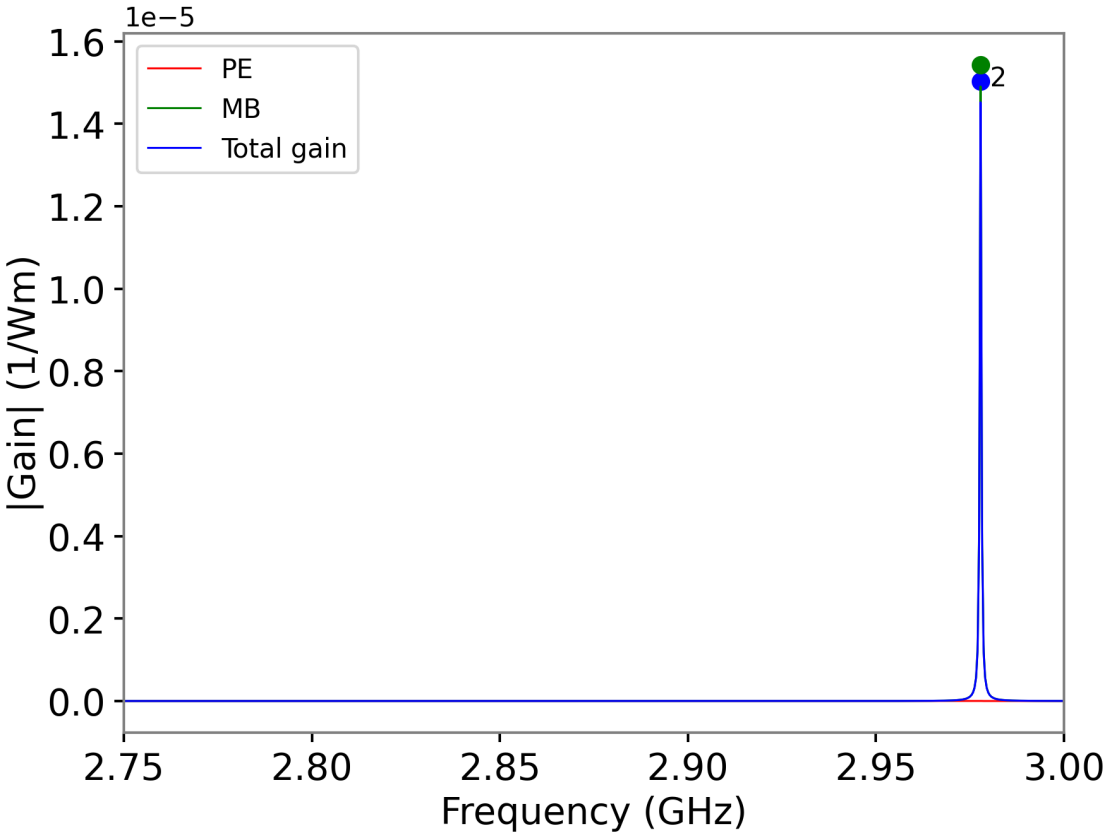



Fig. 19: Gain spectrum for intermodal forward SBS of the silicon waveguide.

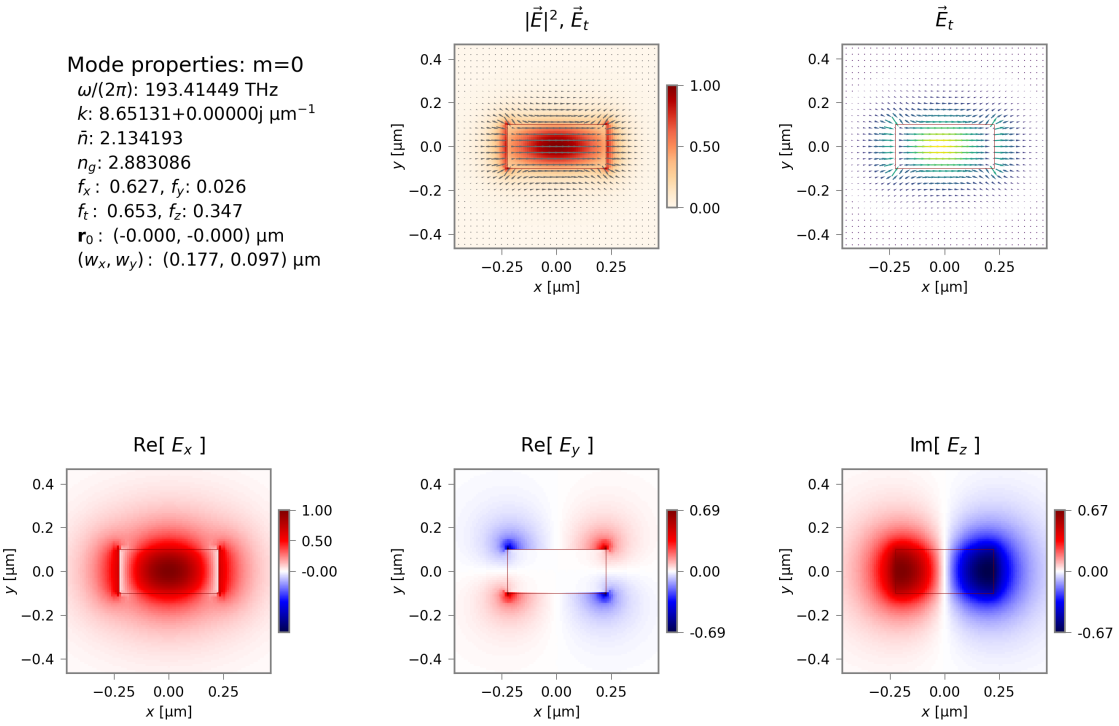



Fig. 20: Fundamental optical mode field.

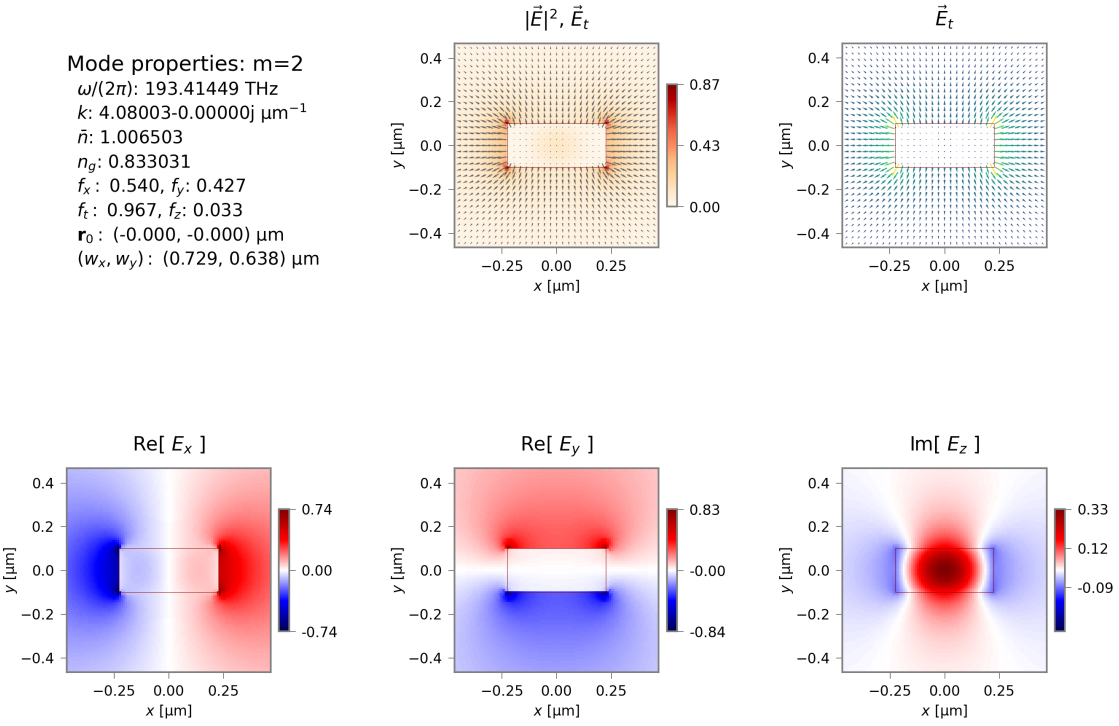



Fig. 21: Second order optical mode field.

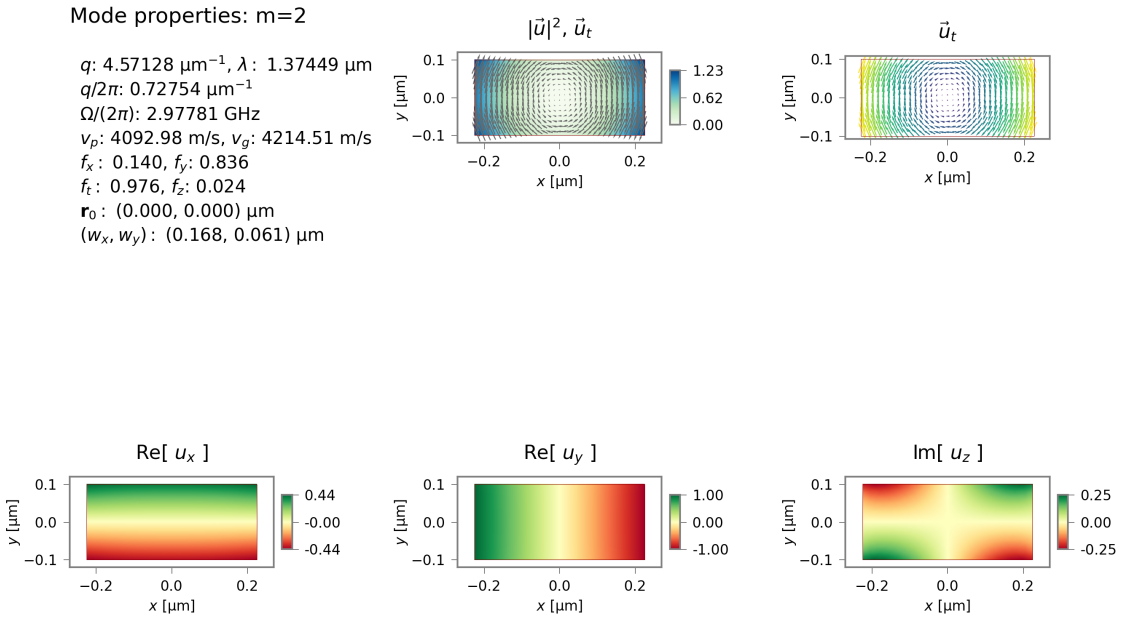



Fig. 22: Elastic mode field of maximum gain.

CHAPTER

TEN

# ADDITIONAL LITERATURE EXAMPLES

Having become somewhat familiar with NumBAT, we now set out to replicate a number of examples from the recent literature located in the `lit_examples` directory. The examples are presented in chronological order. We note the particular importance of examples 5-8 which include experimental and numerical results that are in good agreement.

## 10.1 Example 1 – BSBS in a silica rectangular waveguide

This example `sim-lit_01-Laude-AIPAdv_2013-silica.py` is based on the calculation of backward SBS in a small rectangular silica waveguide described in V. Laude and J.-C. Beugnot, Generation of phonons from electrostriction in small-core optical waveguides, *AIP Advances* **3**, 042109 (2013).

Observe in the python simulation file, the use of a material named `SiO2_2013_Laude` specifically modelled on the parameters in this paper. This naming scheme for materials allows several different versions of nominally the same material to be selected, so that users can easily compare calculations to other authors' exact parameter choices, without changing their preferred material values for their own samples and experiments.

In this paper, Laude and Beugnot plot a spectrum of the elastic energy density, which is not directly measureable. However the spectral peaks show close alignment with the NumBAT gain spectrum for the same structure, as is apparent from the first two plots below. We attribute the remaining difference in the location of the spectral peaks to the choice of elastic material properties in the paper. The paper reports a bulk shear velocity of 3400 m/s, which is around 8% smaller than the usual value of approximately 3760 m/s, but the full set of material properties used in the calculations are not provided. Hence we have used a more standard set of values for silica.

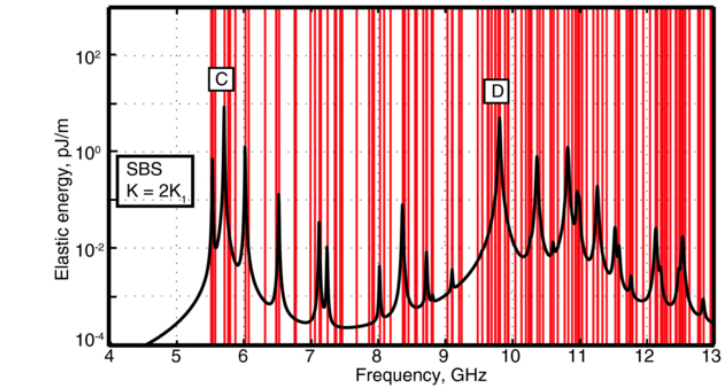



Fig. 1: Spectrum of elastic mode energy calculated in Laude and Beugnot for backward SBS in a silica waveguide.

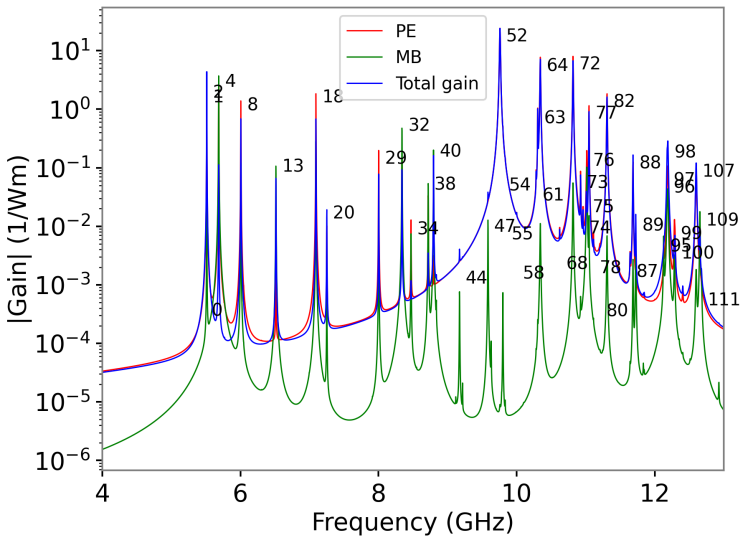



Fig. 2: NumBAT gain spectrum on logy axis.

The optical fundamental fields are in close agreement with values for the effective index of $n_{\text{eff}} = 1.2710$ (NumBAT) and 1.2748 (paper).

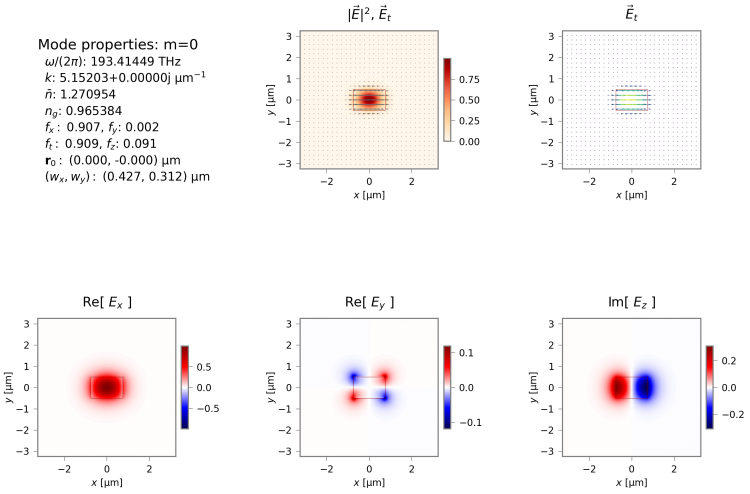



Fig. 3: Fundamental optical mode profiles calculated in NumBAT.

The next set of plots looks at the modal profiles for the for the peaks marked C and D in the first spectrum above. The most direct comparison comes from looking at the NumBAT contour plots for the transverse ($u_x$ and $u_y$) and axial/longitudinal ($u_z$) elastic fields. Note that the transverse and axial plots for mode D in the paper are inconsistent. The axial plot is the correct one for this mode.

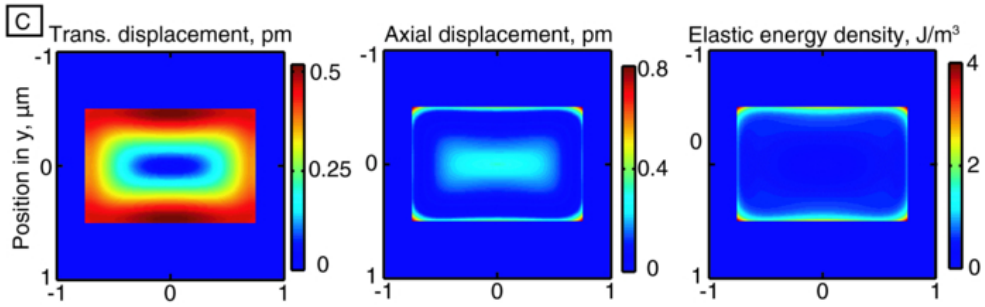



Fig. 4: Elastic mode profiles calculated in Laude and Beugnot for backward SBS in a silica waveguide of diameter 1.05 micron. Mode C corresponds to the peak marked in the spectrum above.

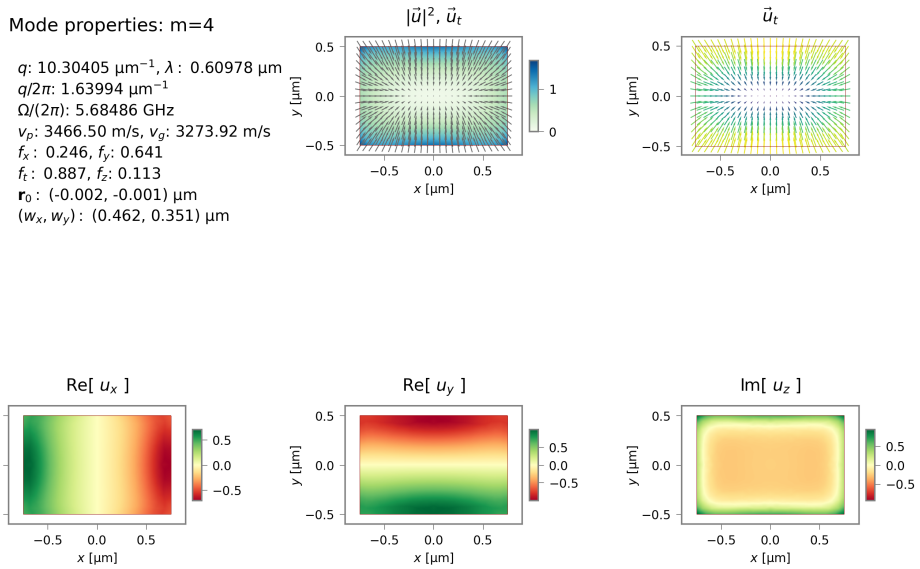



Fig. 5: NumBAT calculation of a high gain elastic mode, marked as C in the paper.

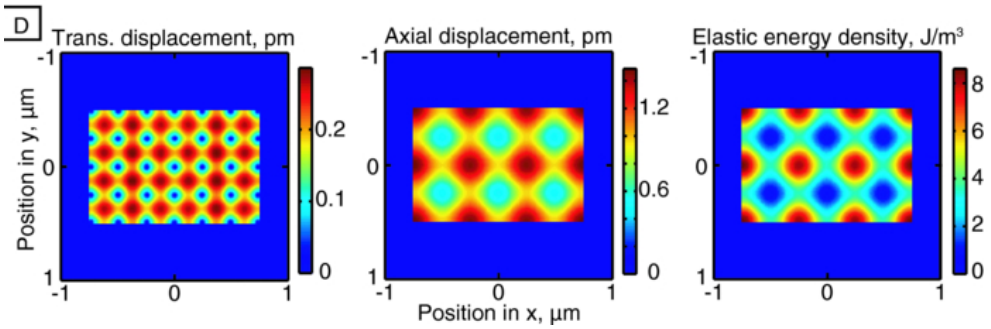



Fig. 6: Elastic mode profiles calculated in Laude and Beugnot for backward SBS in a silica waveguide of diameter 1.05 micron. Mode D corresponds to the peak marked in the spectrum above.

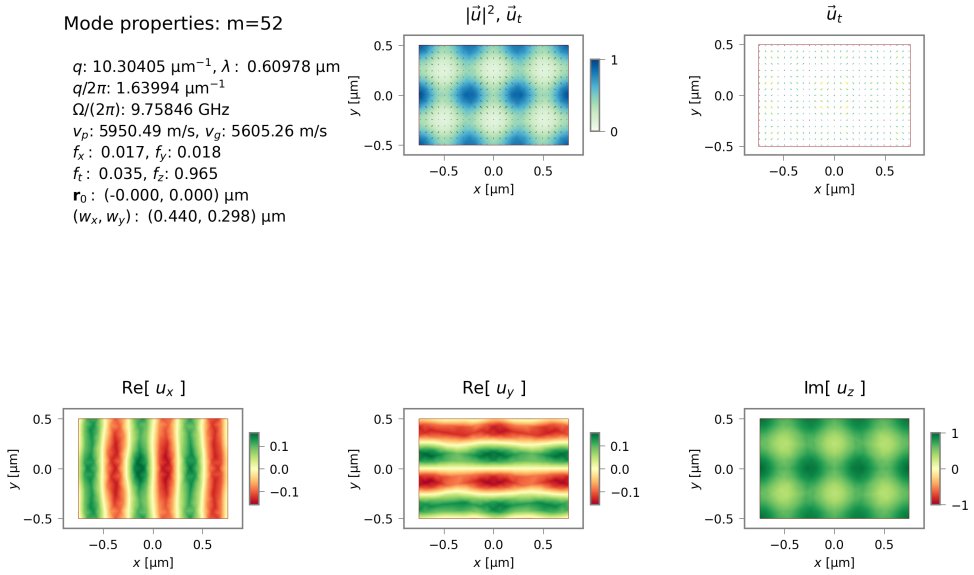



Fig. 7: NumBAT calculation of a high gain elastic mode, marked as D in the paper.

## 10.2 Example 2 – BSBS in a rectangular silicon waveguide

This example in `sim-lit_02-Laude-AIPAdv_2013-silicon.py` again follows the paper of V. Laude and J.-C. Beugnot, Generation of phonons from electrostriction in small-core optical waveguides, *AIP Advances* **3**, 042109 (2013), but this time looks at the *silicon* waveguide case.

Once again, the plots from the original paper are shown in the first figure. In this case, with a very large number of modes of different shear wave order, the precise mode profile for highest gain depends very sensitively on the simulation parameters.

In this case the effective index for the optical fundamental mode are $n_{\text{eff}} = 3.3650$ (NumBAT) and 3.3729 (paper).

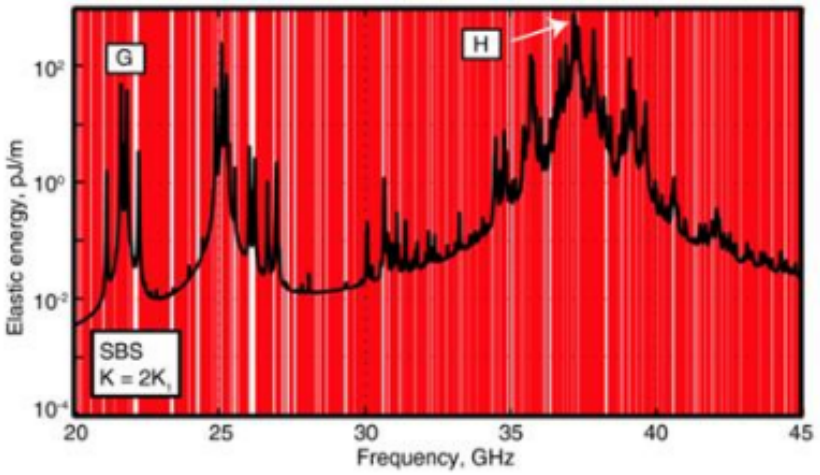



Fig. 8: Spectrum of elastic mode energy calculated in Laude and Beugnot for backward SBS in a silicon waveguide.

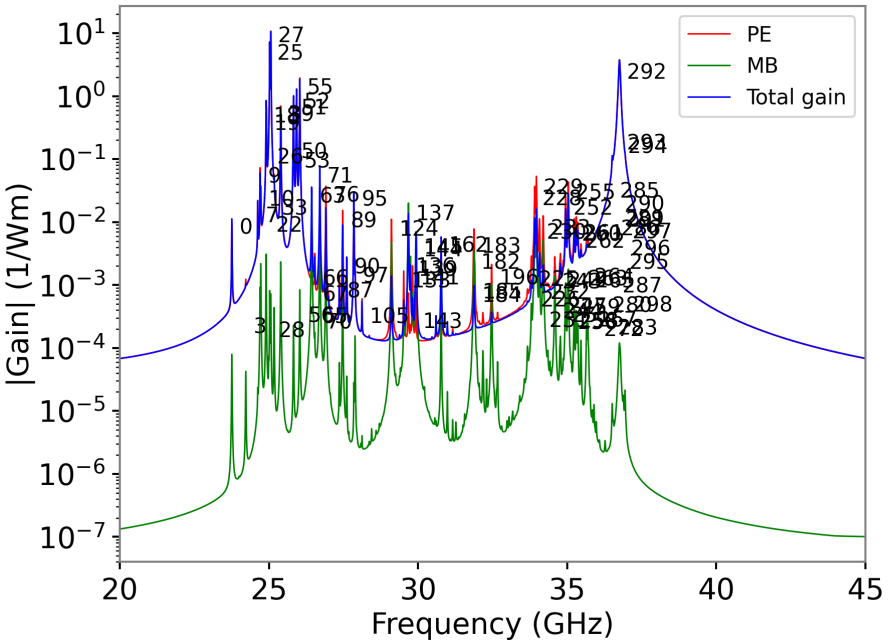



Fig. 9: NumBAT gain spectrum on logy axis.

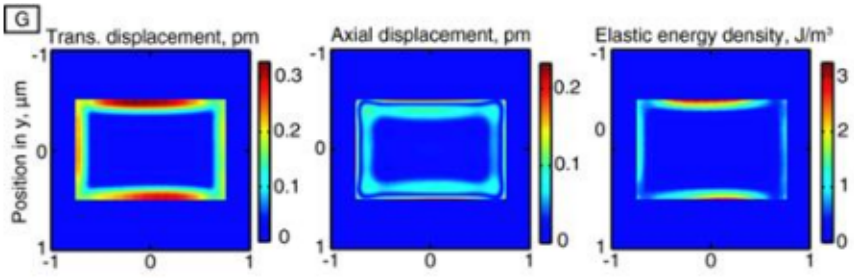



Fig. 10: Field profiles for mode G calculated in Laude and Beugnot for backward SBS in a silicon waveguide.

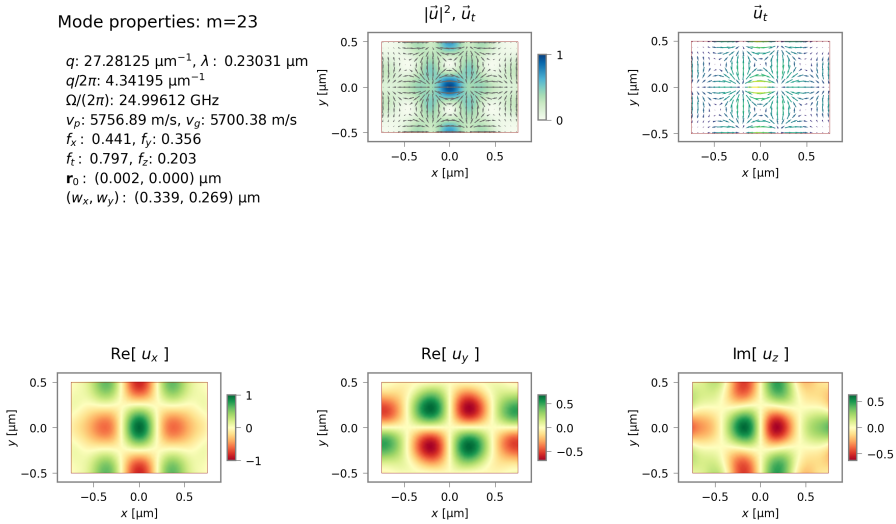



Fig. 11: High gain elastic mode calculated by NumBAT, marked as G in paper.

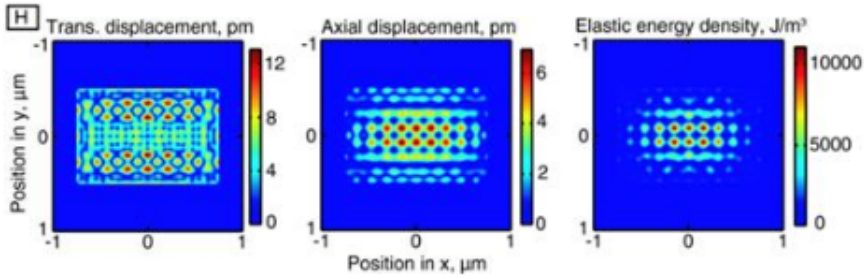



Fig. 12: Field profiles for mode H calculated in Laude and Beugnot for backward SBS in a silicon waveguide.

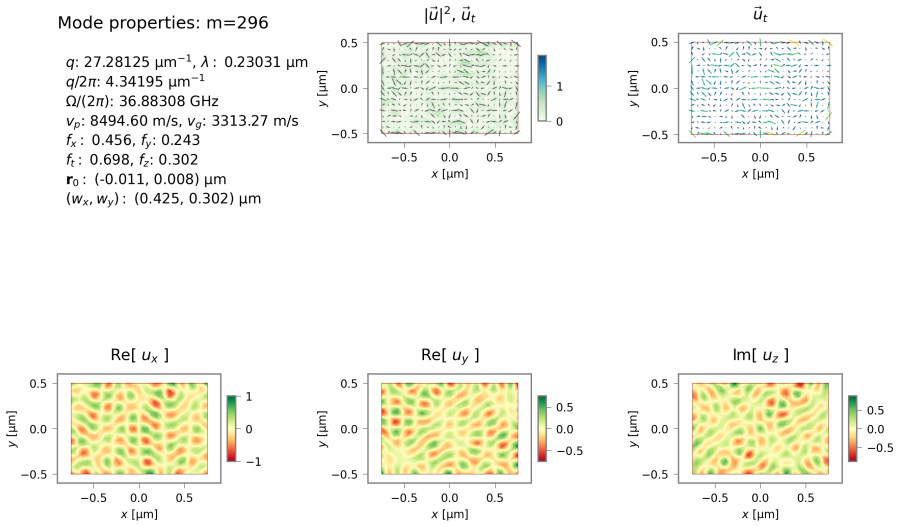



Fig. 13: High gain elastic mode calculated by NumBAT, marked as H in paper.

THINGS TO DO

1. explain the narrower acoustic spectrum for the silicon waveguide.
2. Try comsol on the these problems
3. **plot full elastic mode dispersion, find actual meaning of the elastic** energy. why is this not scaled away by mode normalisation? is this connected to the losses being included in the FEM calc?

## 10.3 Example 3 – BSBS in a tapered fibre - scanning widths

This example, in `sim-lit_03-Beugnot-NatComm_2014.py`, is based on the calculation of backward SBS in a micron scale optical fibre described in J.-C. Beugnot *et al.*, Brillouin light scattering from surface elastic waves in a subwavelength-diameter optical fibre, *Nature Communications* **5**, 5242 (2014).

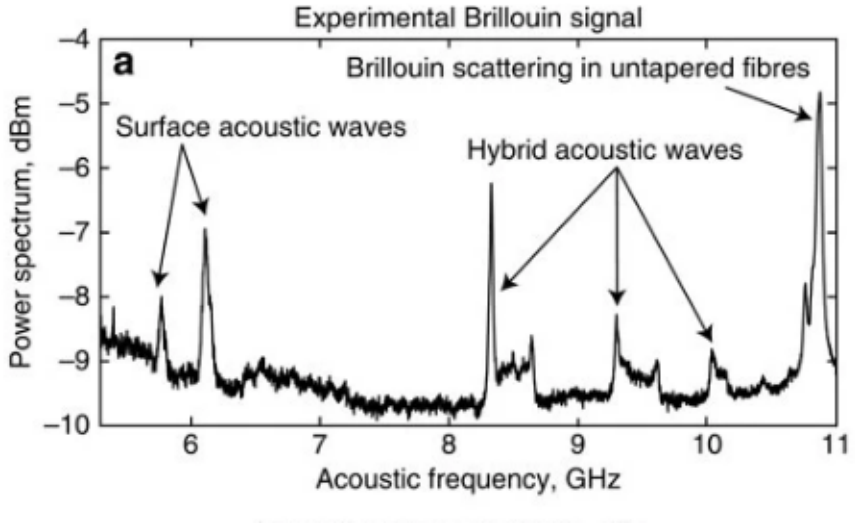



Fig. 14: Measured gain spectrum for a 1.05 micron silica nanowire in J.-C. Beugnot *et al.*

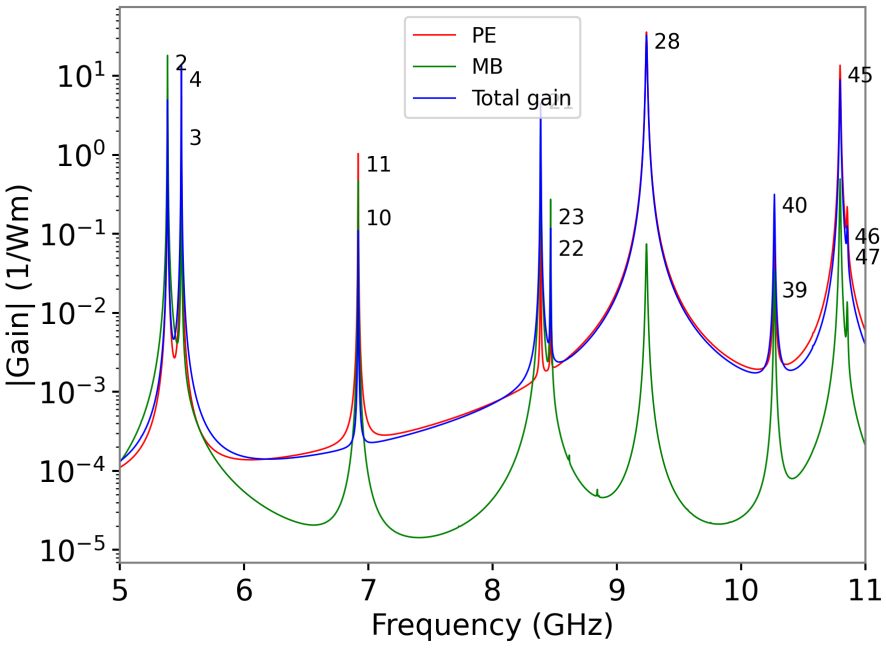



Fig. 15: NumBAT calculated gain spectrum for the 1.05 micron silica nanowire.

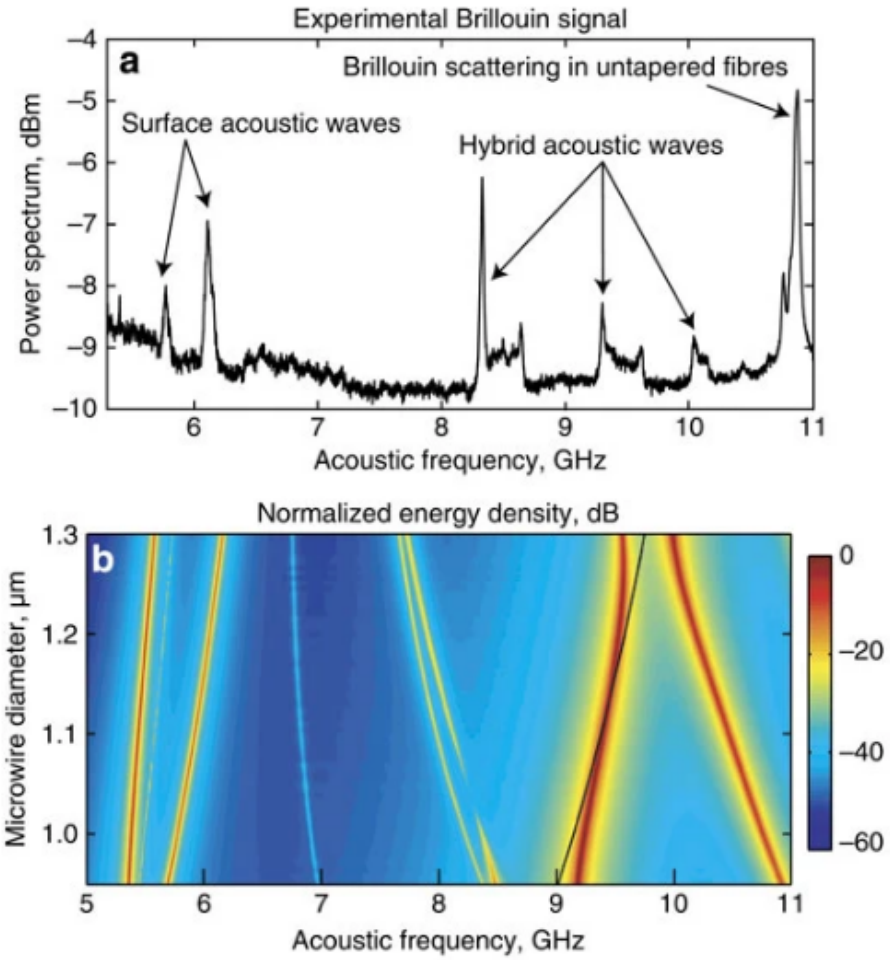



Fig. 16: Calculated dispersion of gain spectrum with nanowire width in J.-C. Beugnot *et al.*

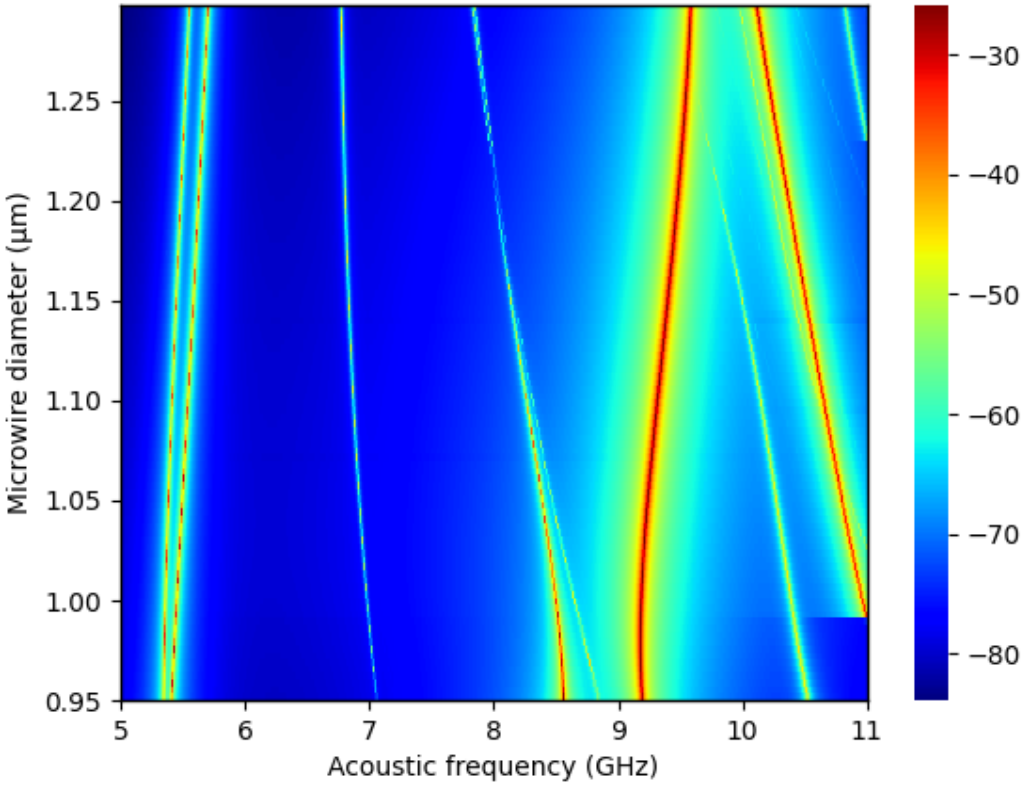



Fig. 17: Full elastic wave spectrum for silica microwire, to be compared with previous plot (Fig. 4a in J.-C. Beugnot *et al.*)

We can now look at the behaviour of some individual modes.

The following table shows the frequency eigenvalues for a number of the modes reported in the paper as calculated by NumBAT and in the article.

Table 1: Comparison of modal frequencies:

| Mode class | Elastic mode number | Beugnot *et al* [GHz] | NumBAT $nu$ [GHz] | Comsol $nu$ [GHz] |
|---|---|---|---|---|
| Fundamental | 0 | _ | 5.18347 | 5.1838 |
| Surface azimuthal | 2 | 5.382 | 5.38522 | 5.3855 |
| Surface radial | 5 | 5.772 | 5.78036 | 5.7807 |
| Hybrid azimuthal | 20 | 8.37 | 8.36492 | 8.3651 |
| Hybrid radial | 28 | 9.235 | 9.24394 | 9.2444 |

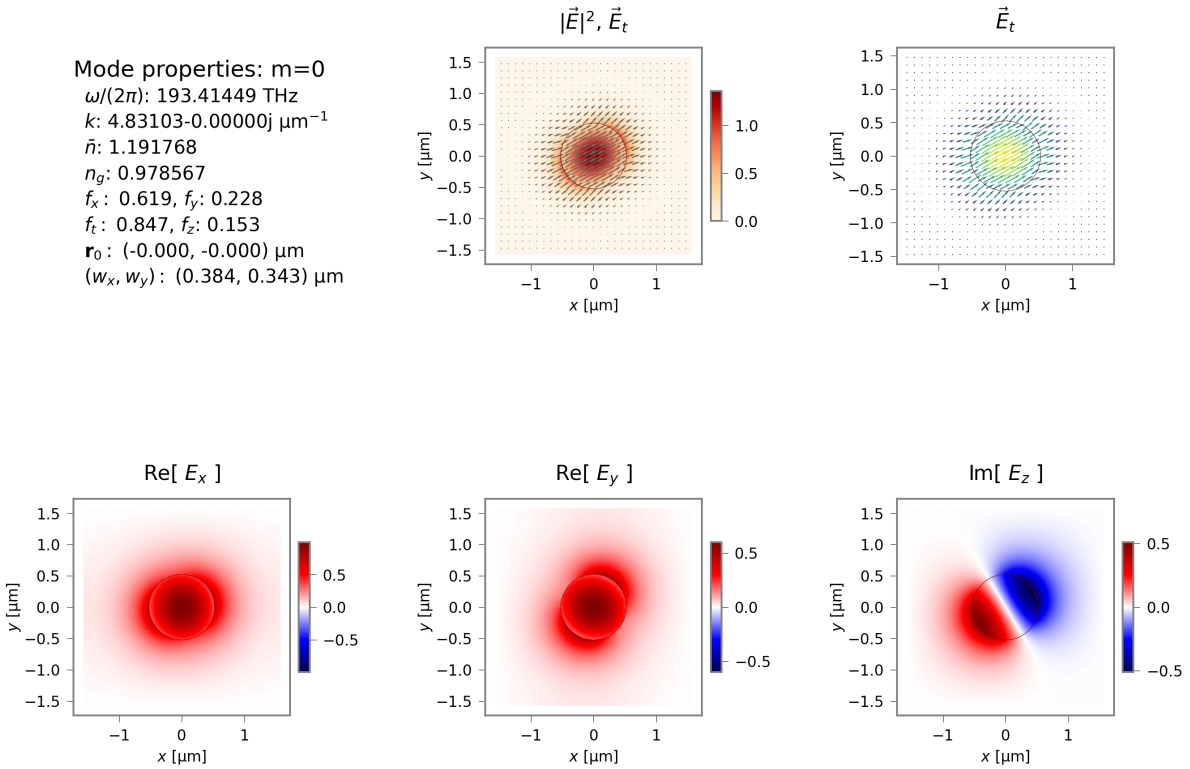



Fig. 18: NumBAT electric field for the fundamental optical mode.

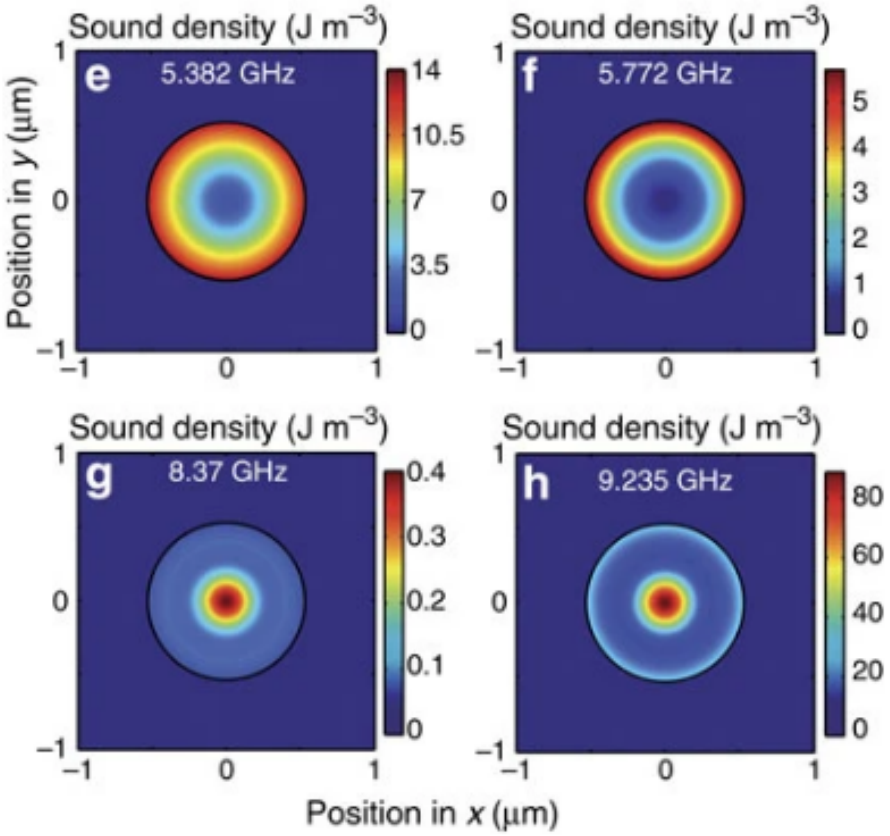



Fig. 19: Calculated elastic mode profiles for the 1.05 micron nanowire in J.-C. Beugnot *et al.* marked in the measured gain spectrum above. The top row are surface modes, the bottom row are hybrid modes.

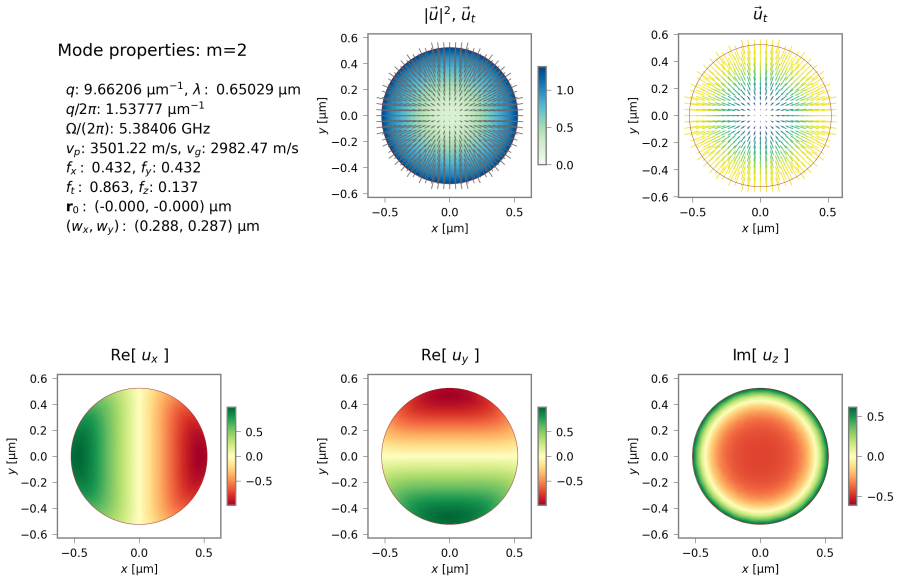



Fig. 20: NumBAT radial shear mode corresponding to the 5.382 GHz surface mode above.

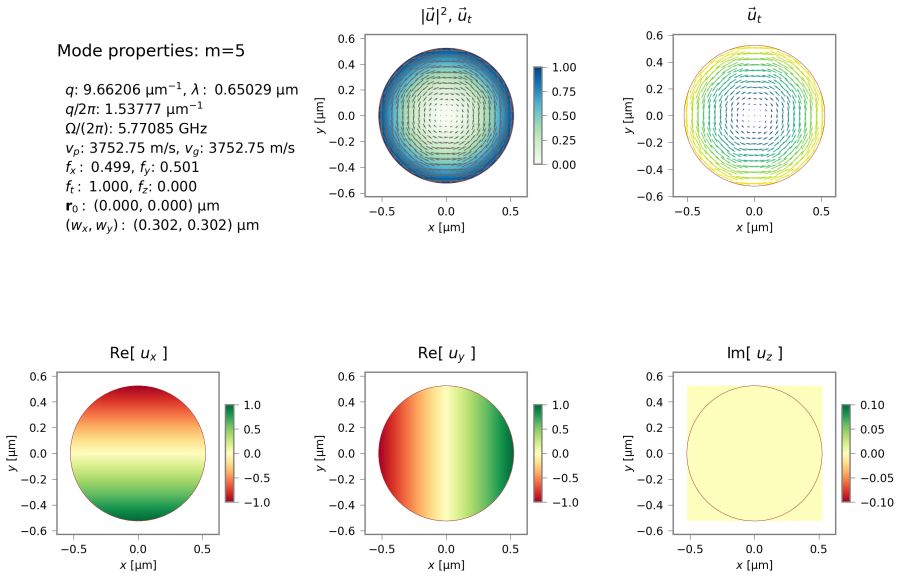



Fig. 21: NumBAT azimuthal shear mode corresponding to the 5.772 GHz surface mode above.

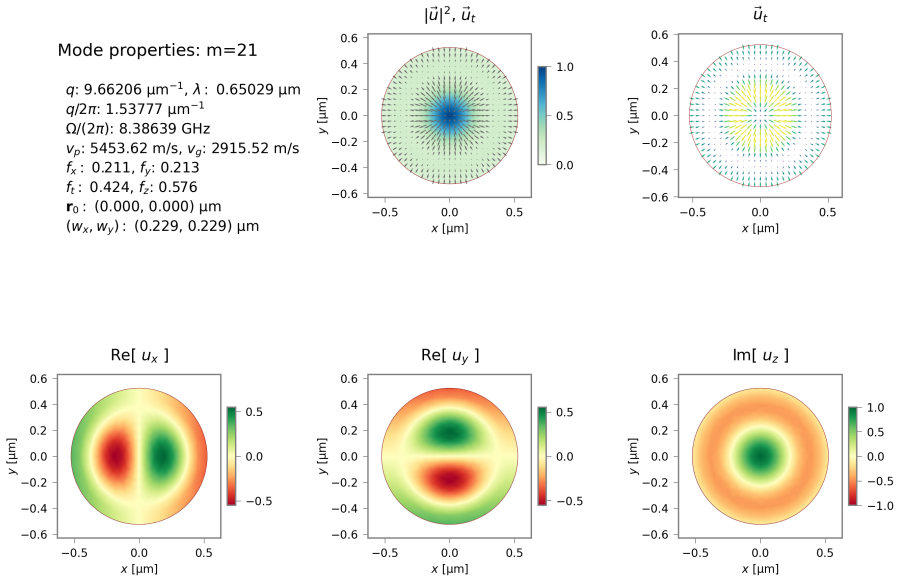



Fig. 22: NumBAT radial shear mode corresponding to the 8.40 GHz surface mode above.

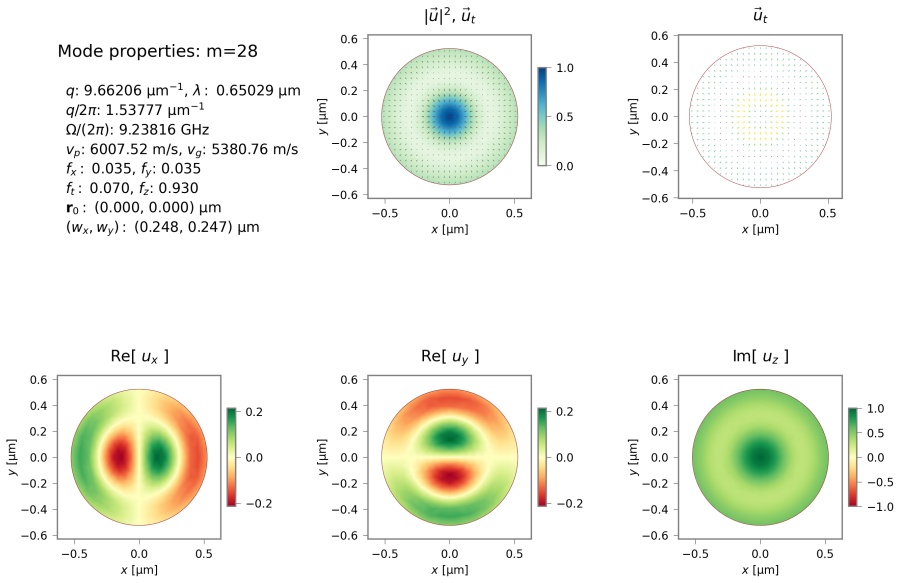



Fig. 23: NumBAT radial shear mode corresponding to the 9.235 GHz surface mode above.

It is also illuminating to look at the elastic dispersion for a wide range of microwire diameters:

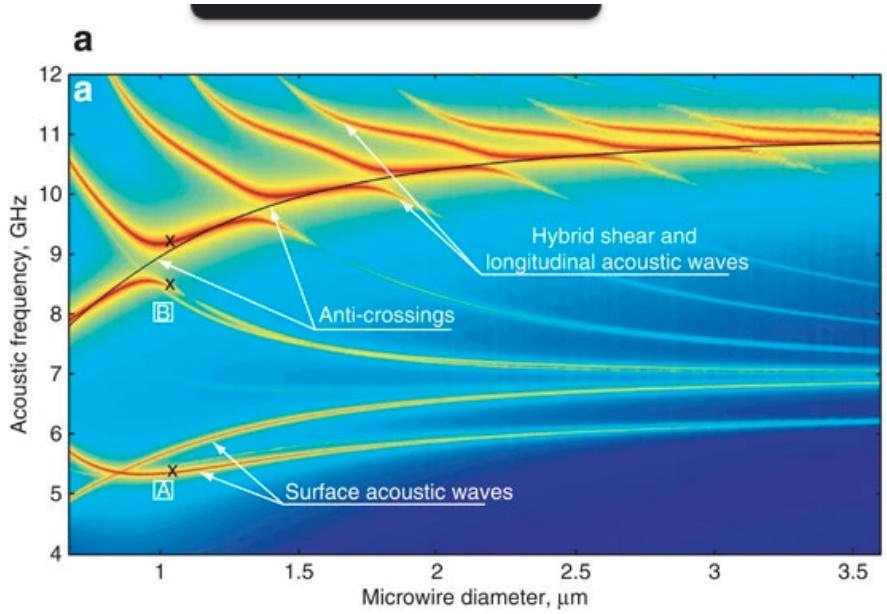



Fig. 24: Elastic wave dispersion over wide range of diameters according to Beugnot *et al.*.

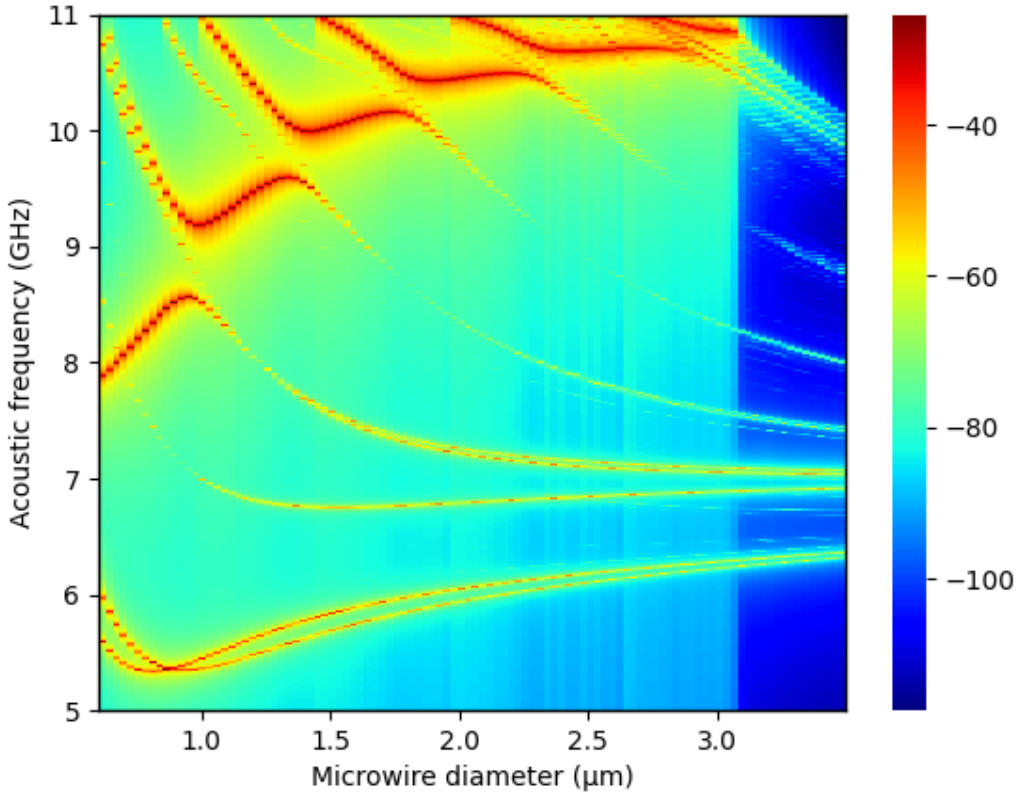



Fig. 25: NumBAT gain spectrum over wide range of diameters. (Note swapped axes for consistency with Beugnot *et al.*.)

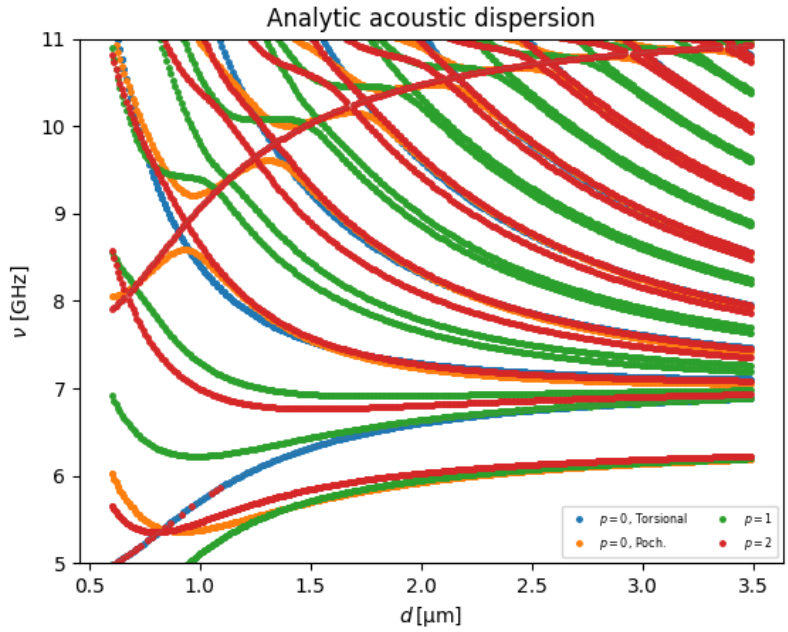



Fig. 26: Exact analytic dispersion relation for modes of elastic rod in air for comparison with gain spectra.

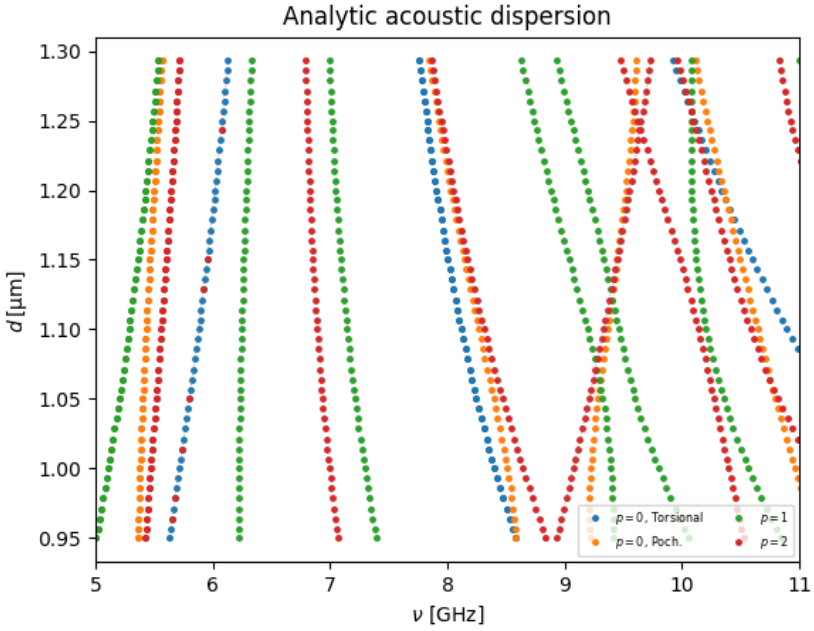



Fig. 27: Exact analytic dispersion relation for modes of elastic rod in air for comparison with gain spectra.

## 10.4 Example 4 – FSBF in a waveguide on a pedestal

This example, in `sim-lit_04-pillar-Van_Laer-NatPhot_2015.py`, is based on the calculation of forward SBS in a pedestal silicon waveguide described in R. Van Laer *et al.*, Interaction between light and highly confined hypersound in a silicon photonic nanowire, *Nature Photonics* **9**, 199 (2015).

Note that the absence of an absorptive boundary in the elastic model causes a problem where the slab layer significantly distorting elastic modes. Adding this feature is a priority for a future release of NumBAT. The following example shows an approximate way to avoid the problem for now.

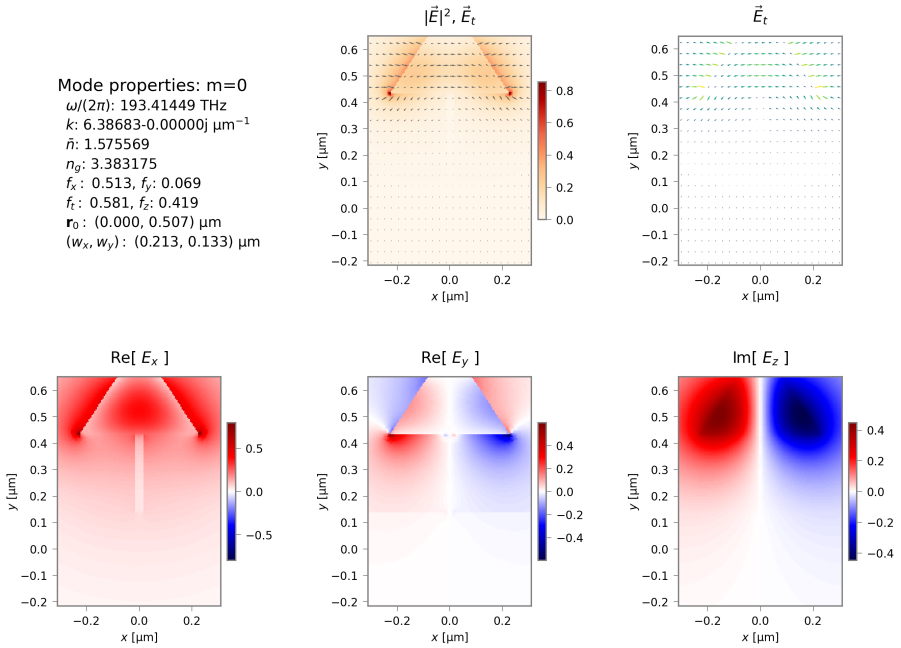



Fig. 28: Fundamental optical mode fields.

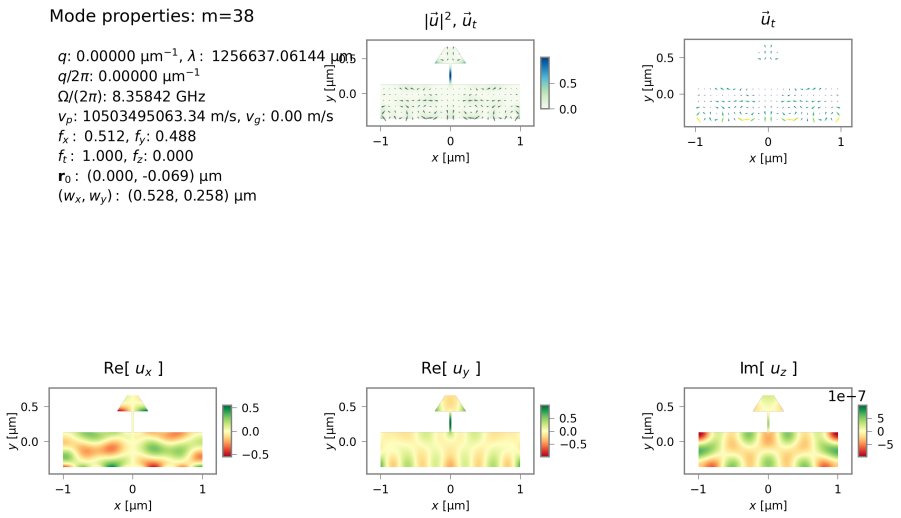



Fig. 29: Dominant high gain elastic mode. Note how the absence of an absorptive boundary on the SiO2 slab causes this layer to significantly distorted the elastic modes.

We may also choose to study the simplified situation where the pedestal is removed, which gives good agreement for the gain spectrum:

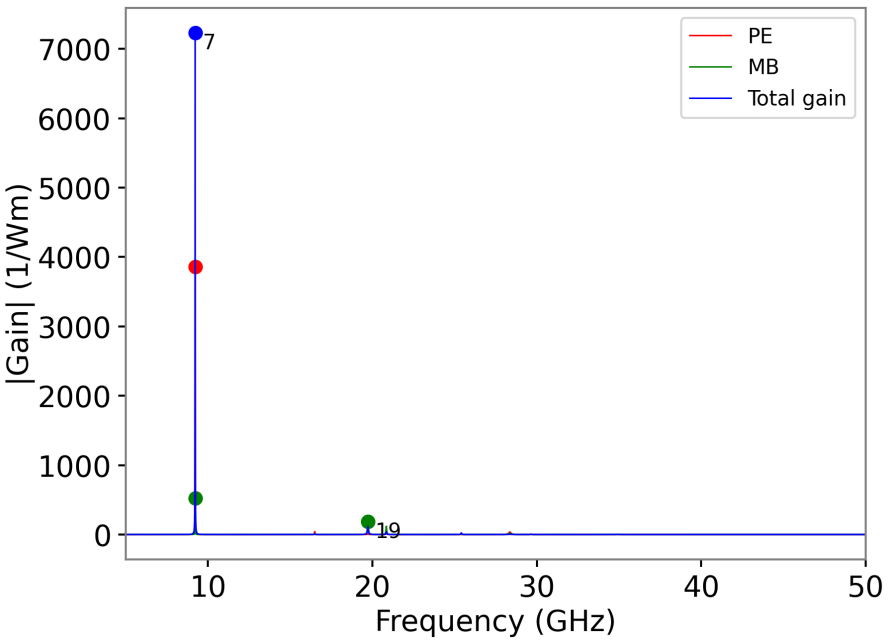



Fig. 30: Gain spectrum for the simplified case of a waveguide surrounded by vacuum.

## 10.5 Example 5 – FSBF in a waveguide without a pedestal

This example, in `sim-lit_05-Van_Laer-NJP_2015.py`, continues the study of forward SBS in a pedestal silicon waveguide described in R. Van Laer *et al.*, Interaction between light and highly confined hypersound in a silicon photonic nanowire, *Nature Photonics* **9**, 199 (2015).

In this case, we simply remove the pedestal and model the main rectangular waveguide. This makes the elastic loss calculation incorrect but avoids the problem of elastic energy being excessively concentrated in the substrate.

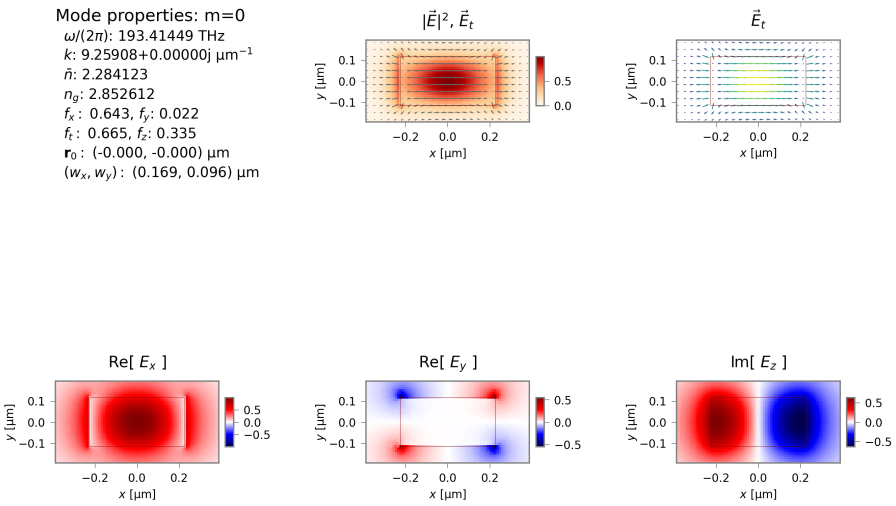



Fig. 31: Fundamental optical mode fields.

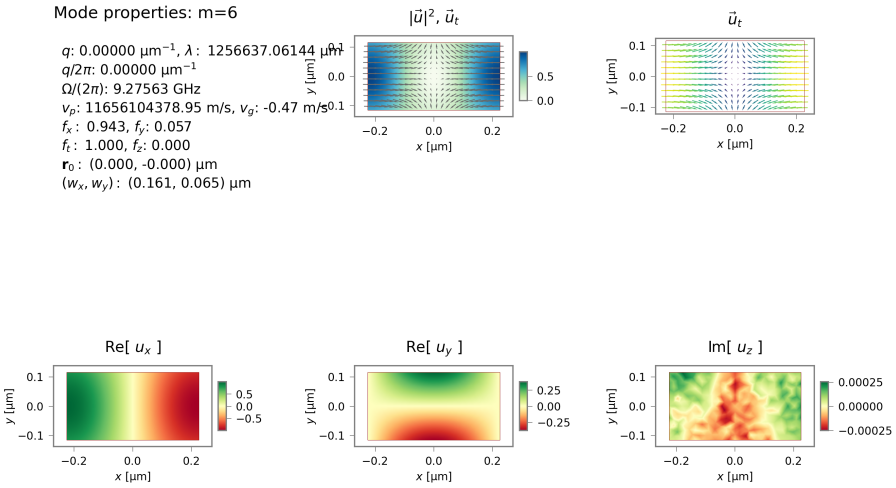



Fig. 32: Dominant high gain elastic mode.

## 10.6 Example 6 – BSBS self-cancellation in a tapered fibre (small fibre)

This example, in `sim-lit_06_1-Florez-NatComm_2016-d550nm.py`, looks at the phenomenon of Brillouin "self-cancellation" due to the electrostrictive and radiation pressure effects acting with opposite sign. This was described in O. Florez *et al.*, Brillouin self-cancellation, *Nature Communications* **7**, 11759 (2016).

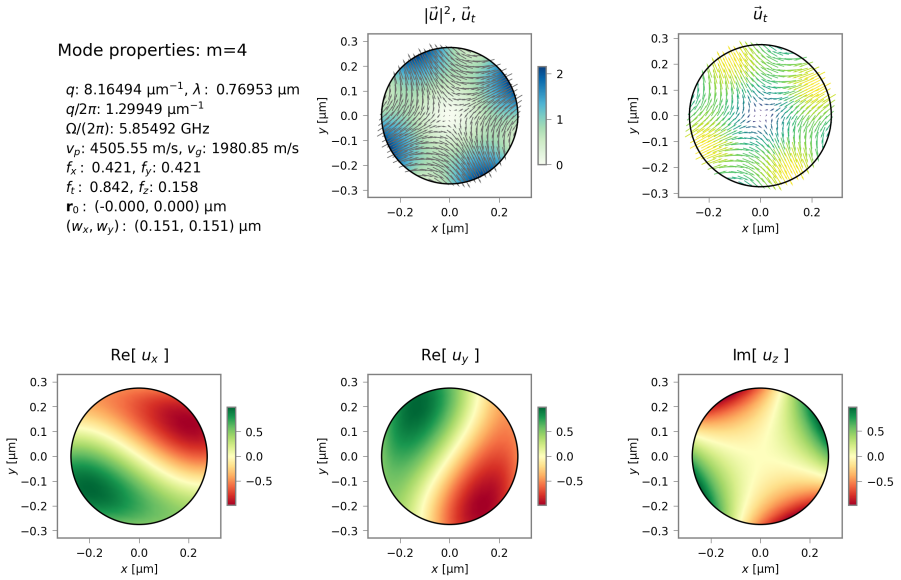



Fig. 33: $TR_{21}$ elastic mode fields of a nanowire with diameter 550 nm.

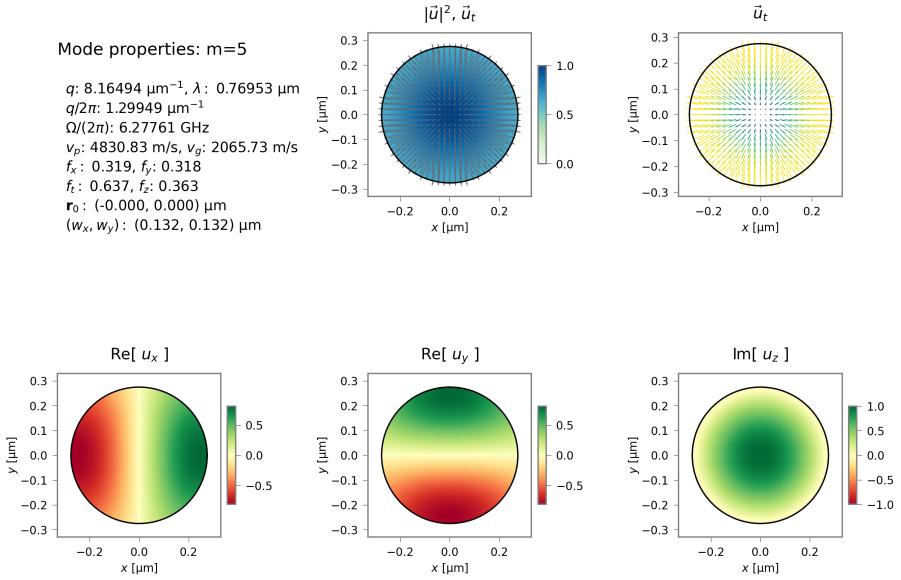



Fig. 34: $R_{01}$ elastic mode fields of a nanowire with diameter 550 nm.

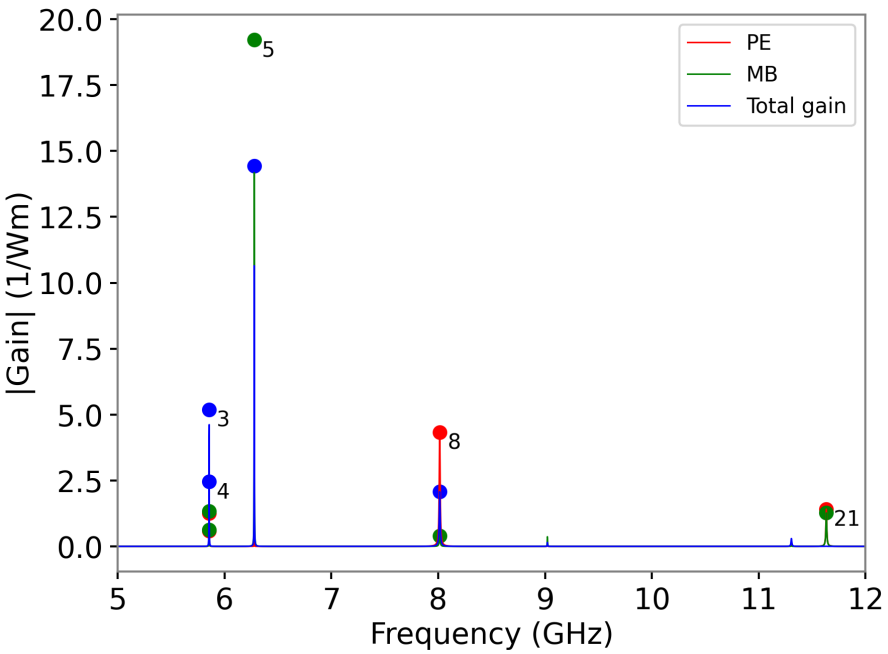
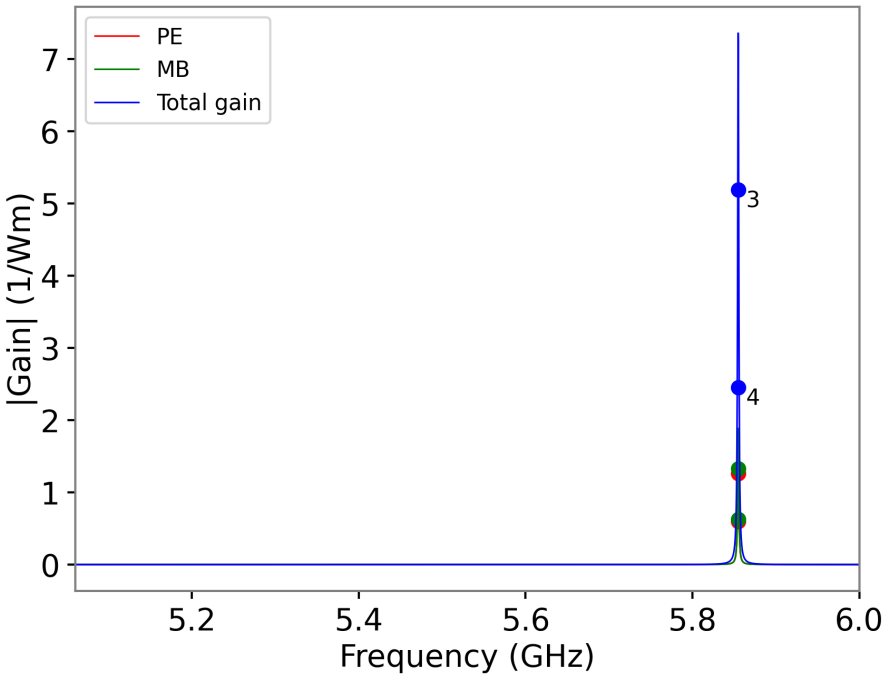
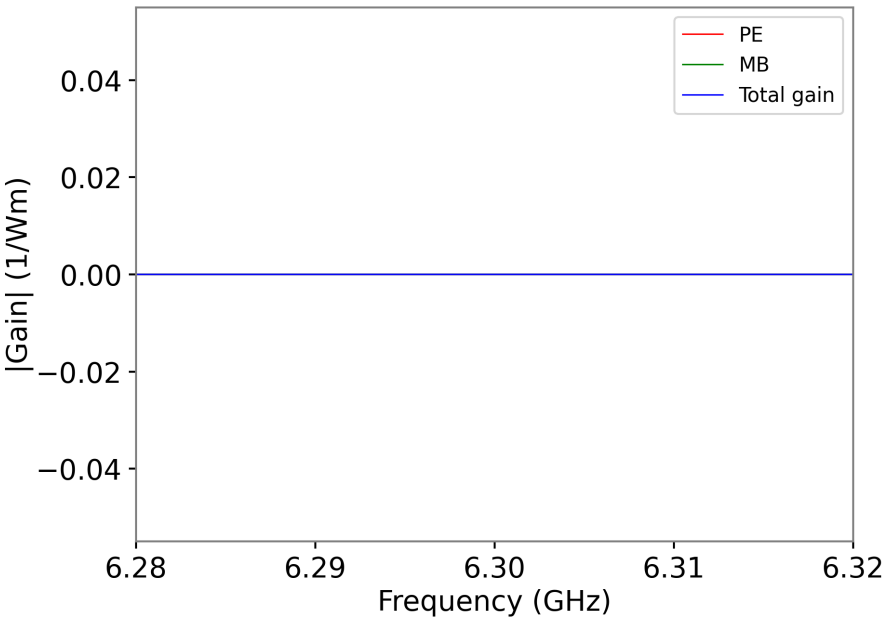
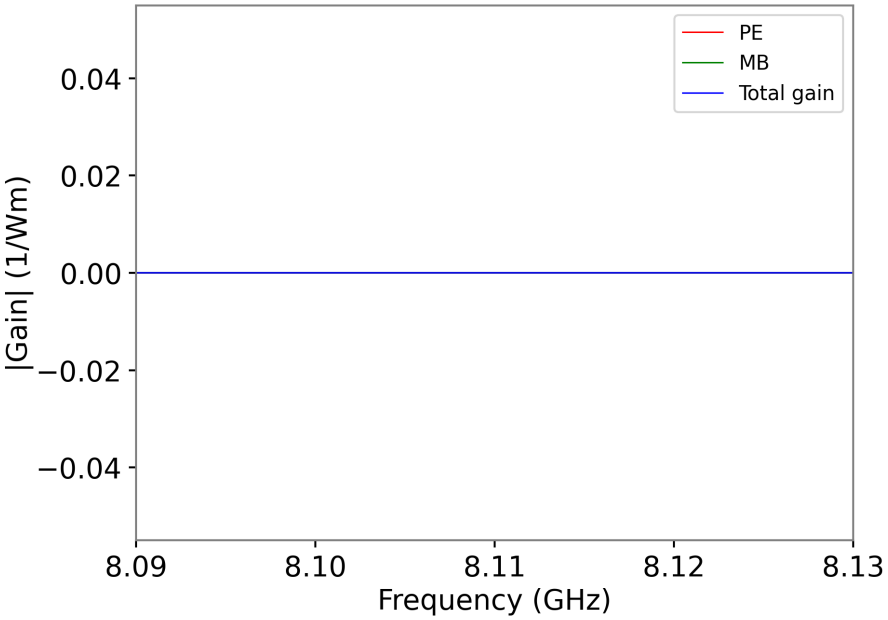



Fig. 35: Gain spectra of a nanowire with diameter 550 nm, matching blue curve of Fig. 3b in paper.




PE
MB
Total gain
0.04
0.02
0.00
−0.02
−0.04
|Gain| (1/Wm)
6.28
6.29
6.30
6.31
6.32
Frequency (GHz)


PE
MB
Total gain
0.04
0.02
0.00
−0.02
−0.04
|Gain| (1/Wm)
8.09
8.10
8.11
8.12
8.13
Frequency (GHz)

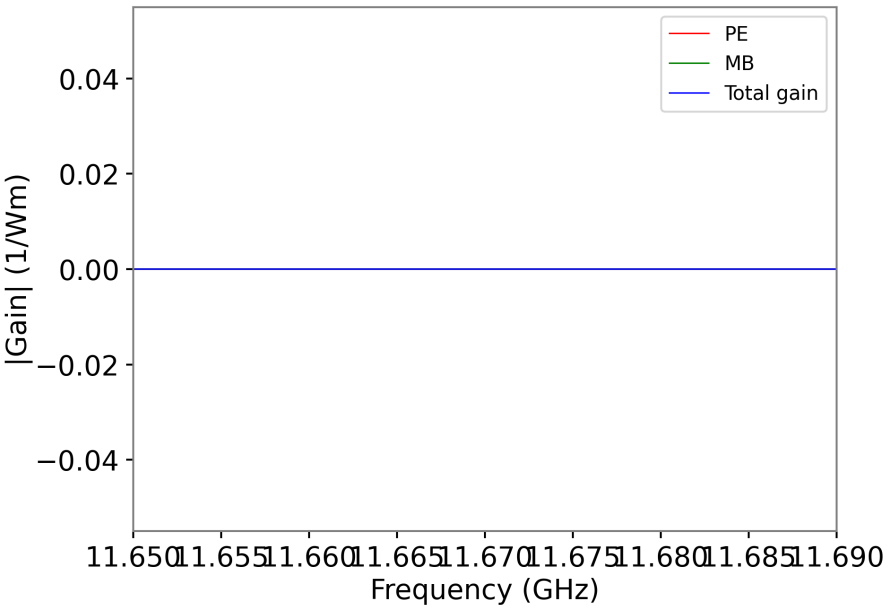



Fig. 36: Zoomed in gain spectra around gain peaks of 550 nm diameter nanowire.

## 10.7 Example 6b – BSBS self-cancellation in a tapered fibre (large fibre)

This example, in `sim-lit_06_2-Florez-NatComm_2016-1160nm.py`, again looks at the paper O. Florez *et al.*, Brillouin self-cancellation, *Nature Communications* **7**, 11759 (2016), but now for a wider core.

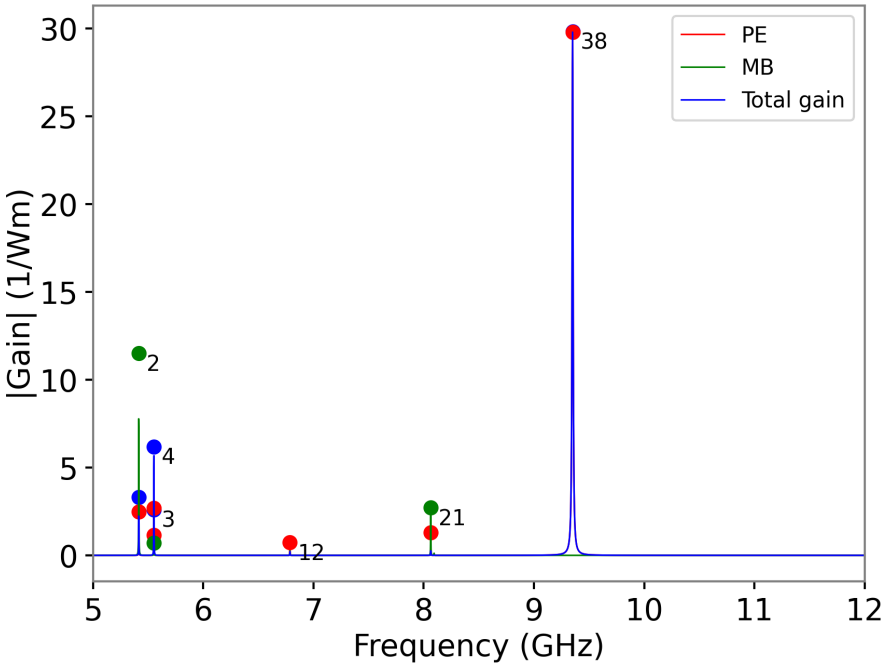



Fig. 37: Gain spectra of a nanowire with diameter 1160 nm, as in Fig. 4 of Florez, showing near perfect cancellation at 5.4 GHz.

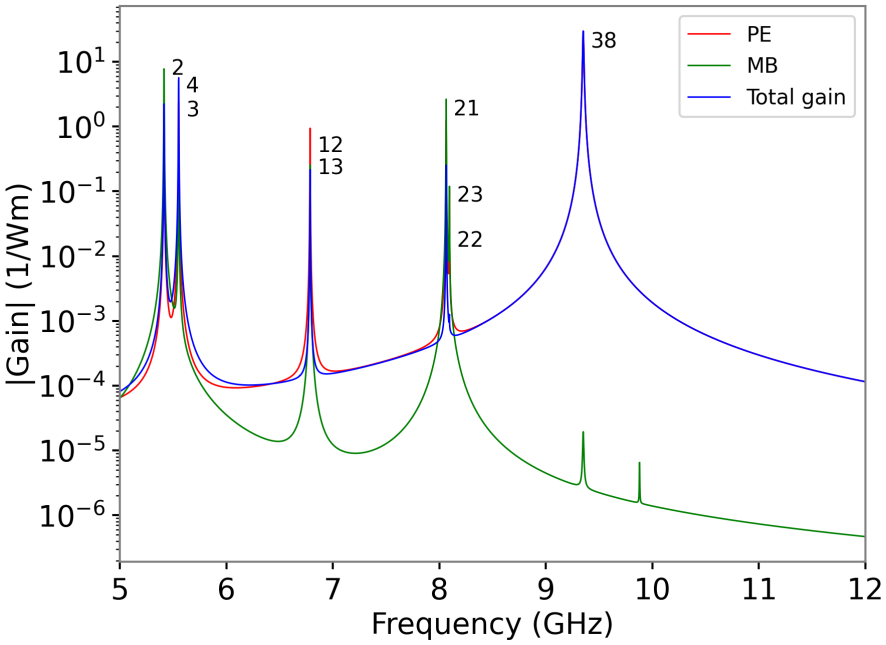



Fig. 38: Gain spectra of a nanowire with diameter 1160 nm, as in Fig. 4 of paper, showing near perfect cancellation at 5.4 GHz.

## 10.8 Example 7 – FSBF in a silicon rib waveguide

This example, in `sim-lit_07-Kittlaus-NatPhot_2016.py`, explores a first geometry showing large forward SBS in silicon as described in E. Kittlaus *et al.*, Large Brillouin amplification in silicon, *Nature Photonics* **10**, 463 (2016).

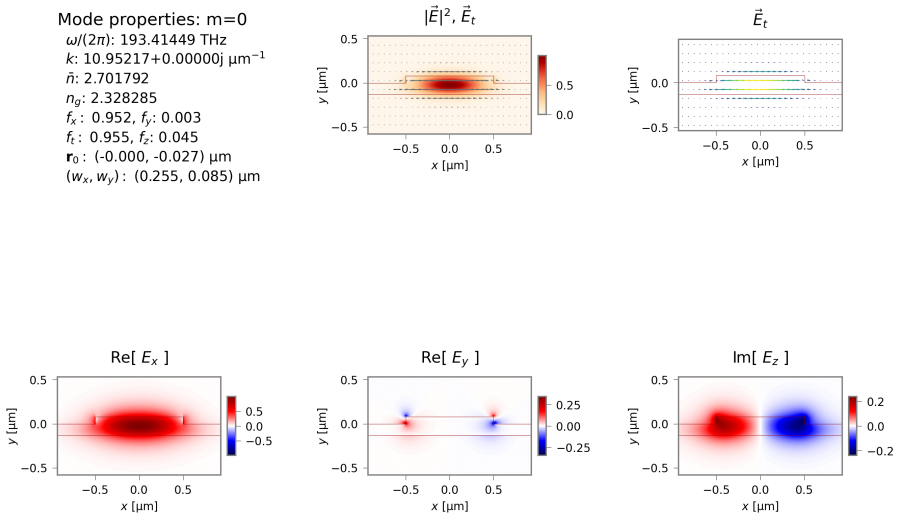



Fig. 39: Fundamental optical mode fields.

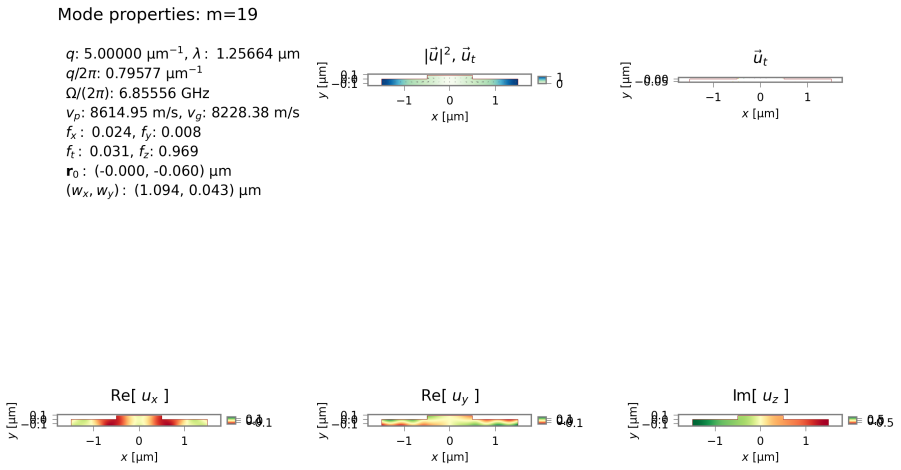



Fig. 40: Dominant high gain elastic mode.

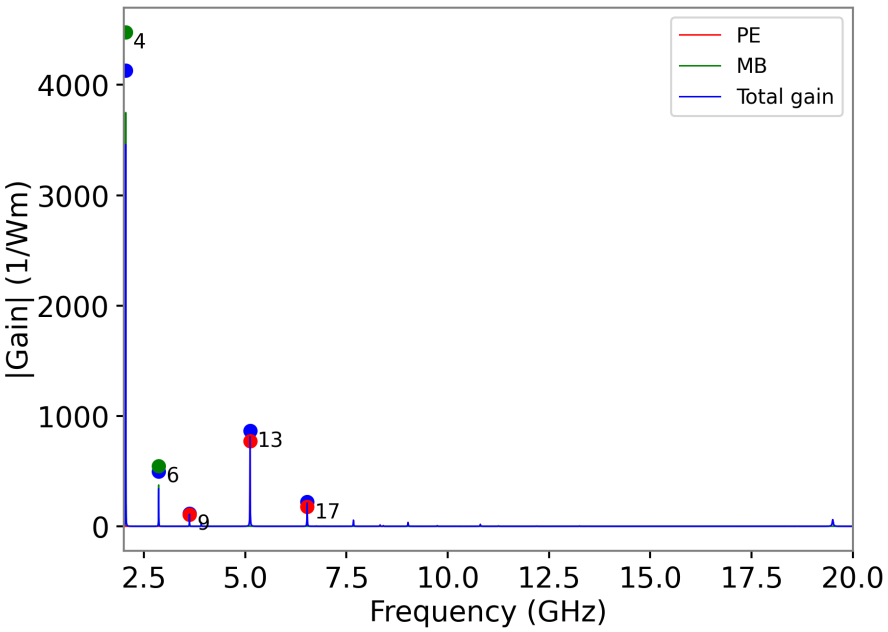



Fig. 41: Gain spectra showing gain due to photoelastic effect, gain due to moving boundary effect, and total gain.

## 10.9 Example 8 – Intermodal FSBF in a silicon waveguide

This example (`sim-lit_08-Kittlaus-NatComm_2017.py`), also from the Yale group, examines intermode forward Brillouin scattering in silicon.

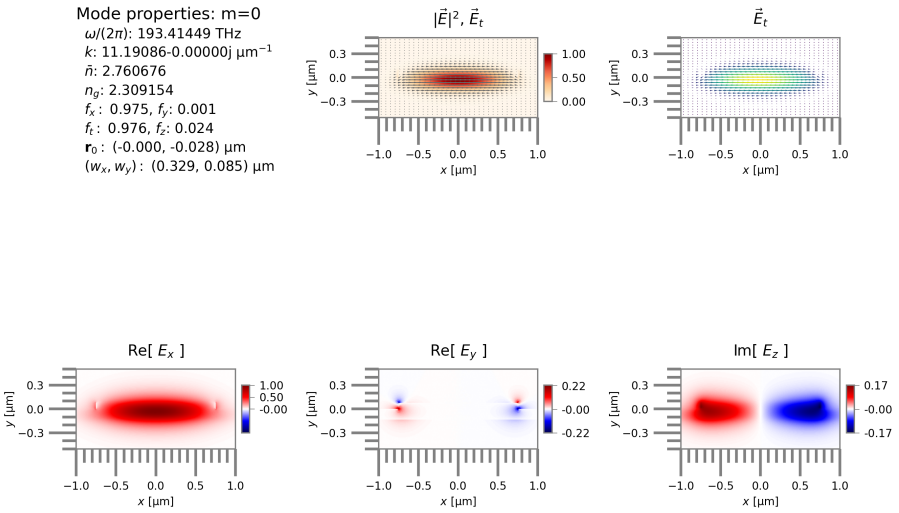



Fig. 42: Fundamental (symmetric TE-like) optical mode fields.

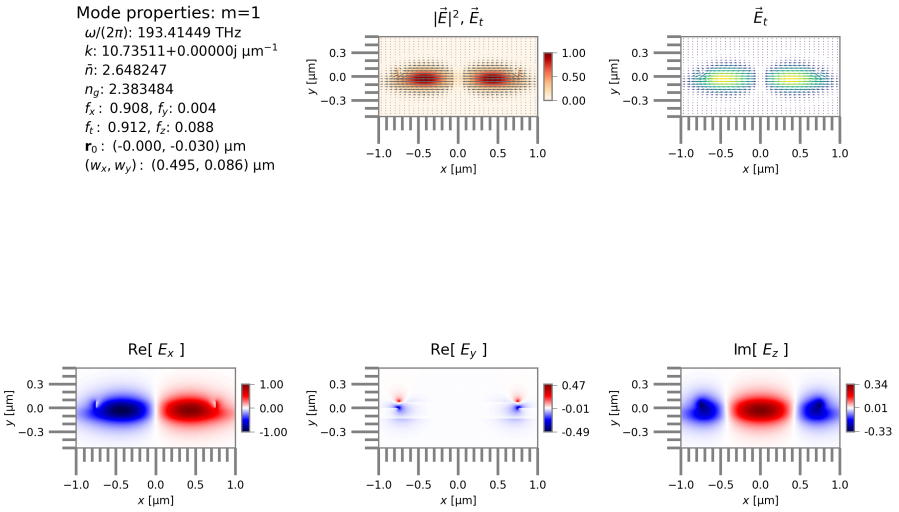



Fig. 43: 2nd lowest order (anti-symmetric TE-like) optical mode fields.

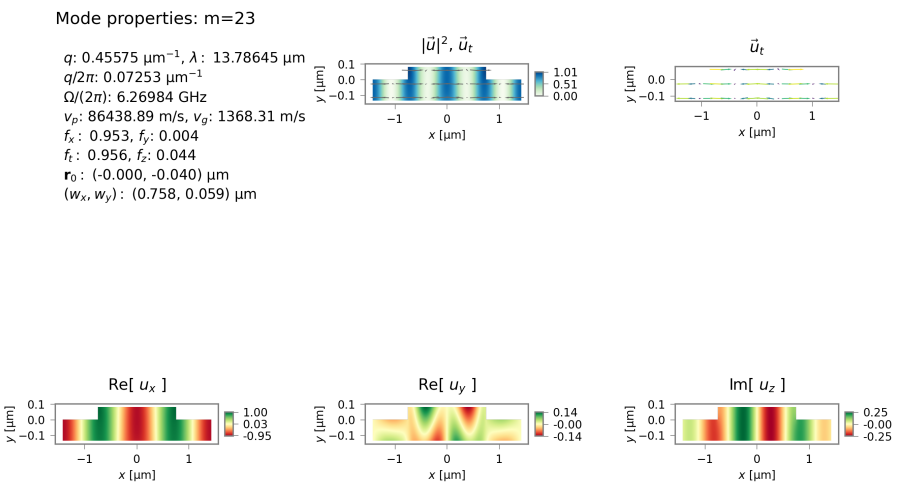



Fig. 44: Dominant high gain elastic mode.

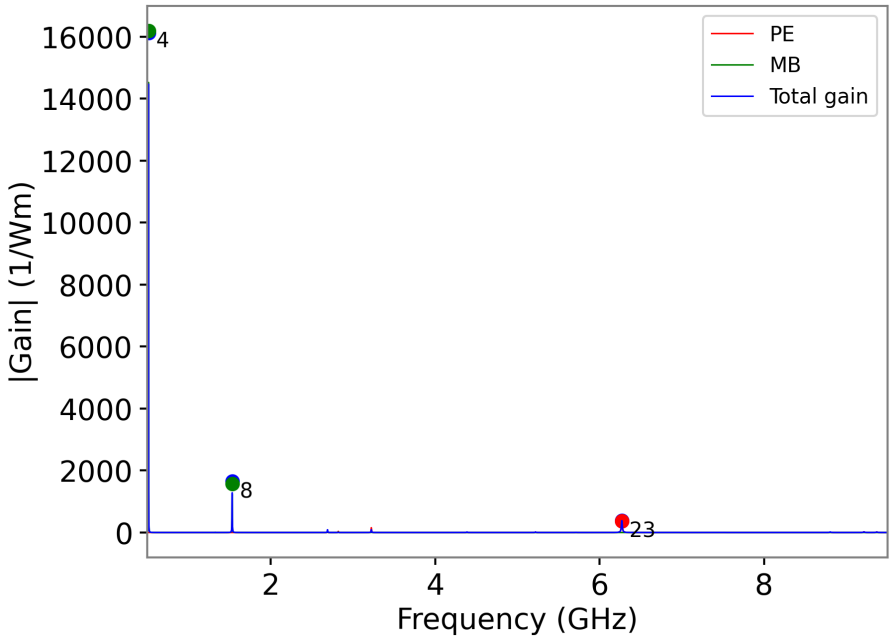



Fig. 45: Gain spectra showing gain due to photoelastic effect, gain due to moving boundary effect, and total gain.

## 10.10 Example 9 – BSBS in a chalcogenide rib waveguide

This example, in `sim-lit_09-Morrison-Optica_2017.py`, from the Sydney group examines backward SBS in a chalcogenide rib waveguide.

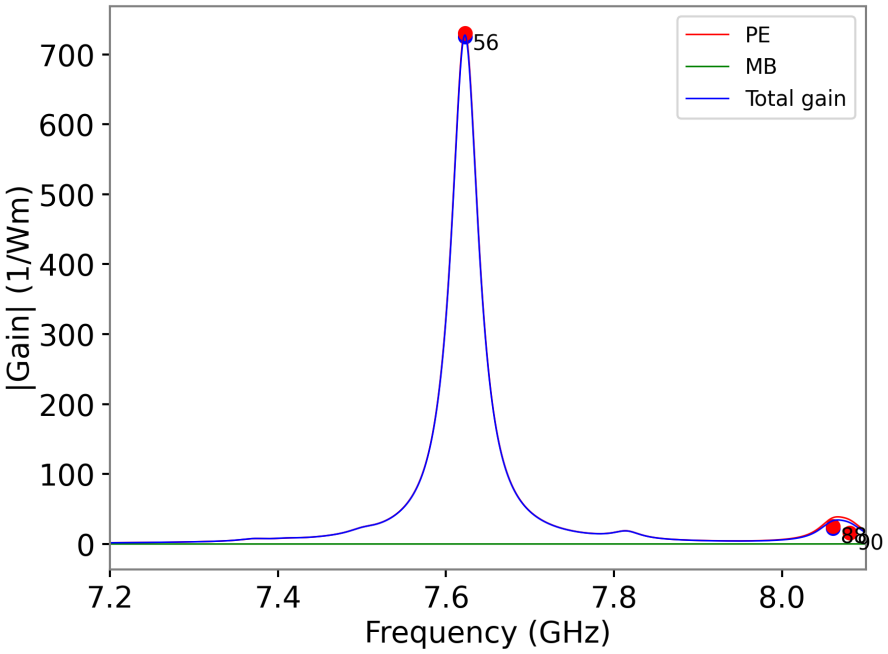
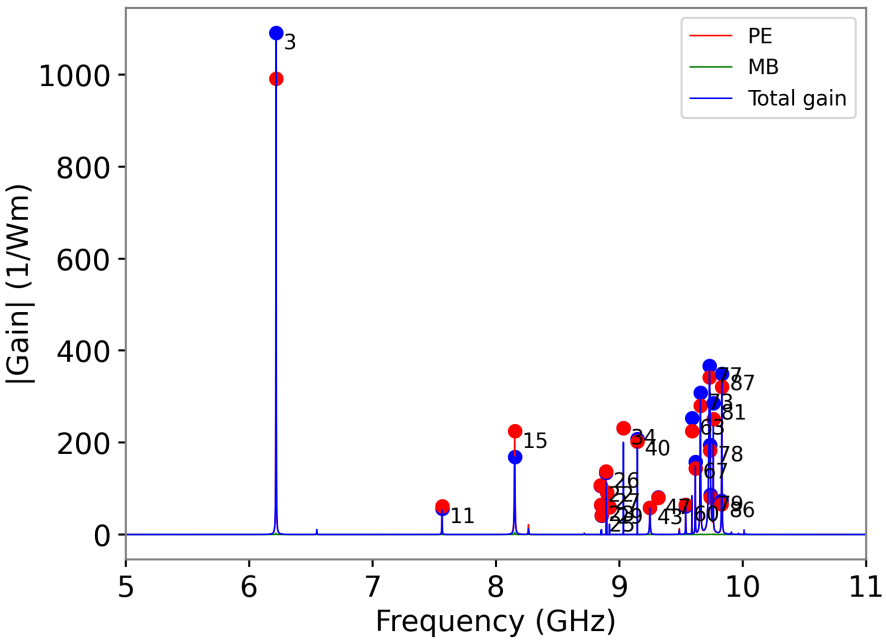


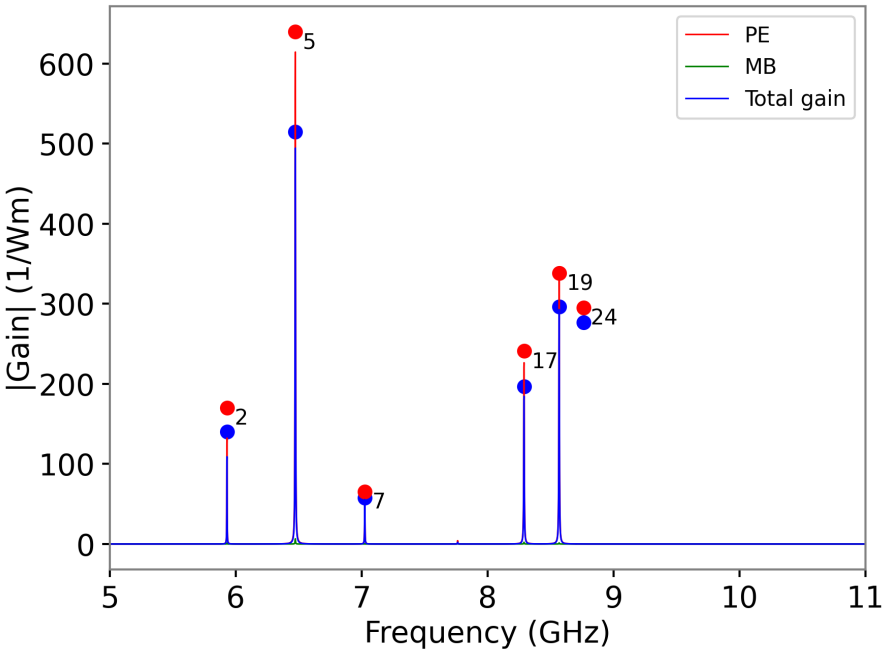

Fig. 46: Gain spectra showing gain due to photoelastic effect, gain due to moving boundary effect, and total gain.

## 10.11 Example 10 – SBS in the mid-infrared

This example, in `sim-lit_10-Wolff-OptExpress-2014.py` and `sim-lit_10a-Wolff-OptExpress-2014.py`, by C. Wolff and collaborators examines backward SBS in the mid-infrared using germanium as the core material in a rectangular waveguide with a silicon nitride cladding.

The second of these two files illustrates the rotation of the core material from the [100] orientation to the [110] orientation. The second file prints out the full elastic properties of the germanium material in both orientations which are seen to match the values in the paper by Wolff et al.


PE
MB
Total gain
|Gain| (1/Wm)
Frequency (GHz)
0
200
400
600
800
1000
5
6
7
8
9
10
11
3
11
15
34
40
63
78
87
81
67
79
86
43
60


PE
MB
Total gain
|Gain| (1/Wm)
Frequency (GHz)
0
100
200
300
400
500
600
5
6
7
8
9
10
11
5
2
7
17
19
24

CHAPTER

ELEVEN

# ADDITIONAL DETAILS

## 11.1 User plot preferences

A large number of plotting properties can be controlled using the `numbat.toml` file.

A sample file is provided in the root directory of the NumBAT installation.

To provide an adjustable set of global settings copy this into your home directory, either as `~/.numbat.toml` on Linux or MacOS, or `numbat.toml` on Windows.

You can also places copies of this file in any working directory to override your normal defaults for a particular set of calculations.

Specifying and using anistropic materials

---

WRITE ME

Orientation of the coordinate axes in NumBAT

---

Cartesian coordinates in NumBAT are defined so that the waveguide lies in the $x-y$ plane with propagation along $z$.

To obtain a right-handed system, one should think of the propagation as being out of the screen (though it is rare that this matters) :

- $x$ : Increases to the right. Usually the dominant electric field component for modes labelled as TE polarisation.
- $y$ : Increases up the screen. Usually the dominant electric field component for modes labelled as TM polarisation.
- $z$ : Increases out of the screen. Propagation direction.

It is common to encounter different coordinate choices in the literature, in particular

the *bird's-eye view* corresponding to viewing a photonic circuit from above is frequently encountered:

- $x$: Increases to the right. Usually the dominant electric field component for modes labelled as TE polarisation.
- $y$: Increases out of the screen. Propagation direction.
- $z$: Increases up the screen. Usually the dominant electric field component for modes labelled as TM polarisation.

When dealing with elastically anisotropic materials, it is important to understand the relationship between the NumBAT coordinate system and that of any literature source you may be consulting, as it may be necessary to perform a rotation on the tensor properties of the material in question. This is discussed below.

### 11.1.1 Supported crystal classes

Depending on the degree of crystal symmetry of a material, the optical and elastic response tensors obey constraints that can substantially reduce the number of independent components.

NumBAT currently supports the following crystal classes: Isotropic, Cubic, Trigonal.

Note that this choice merely reflects what has been useful to the authors and new classes are easily added as needed. Please get in touch.

The number and location of the independent elements of each tensor — stiffness tensor $c_{ijkl}$,

photolelastic tensor $p_{ijkl}$,

viscosity tensor $p_{ijkl}$ — depends on the particular crystal class.

As all the relevant tensors are symmetric, there can be at most 6 independent elements for rank-2 tensors (rather than 9), and 36 independent elements in rank-4 tensors (rather than 81). In particular, tensor subscripts

are all symmetric in pairs as follows:

$$T_{ij} = T_{ji} T_{ijkl} = T_{jikl} = T_{ijlk} = T_{jikl}$$

where the subscripts range over all values of

$x, y, z$.

Consequently we can use the Voigt notation

in which pairs of indices are represented by a single integer(8) in the range 1 to 6 as follows:

$$\begin{bmatrix} xx \\ yy \\ zz \\ xz \\ yz \\ zz \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}$$

Then a rank two tensor like strain or stress can be represented as the column

$$\bar{S} \equiv S_I \equiv \begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \end{bmatrix}.$$

A fourth rank tensor like the stiffness or photoelastic tensor is represented in the form

$$\bar{c} \equiv c_{IJ} \equiv \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} \end{bmatrix}.$$

This table summarises the symmetry properties for each class and the required tensor elements that must be specified in a material `.json` file.

Table 1: Properties of crystal classes and required elements in NumBAT.

| Structure | Nature | Examples | $\epsilon$ | Stiffness elements | Photoelastic elements |
|---|---|---|---|---|---|
| Isotropic | Every direction equivalent | Glass | $\epsilon_1$ | $c_{11}$ | $p_{11}, p_{12}, p_{14}$ |
| Cubic | 3 equivalent perpendicular directions | Silicon | $\epsilon_1$ | $c_{11}, c_{12}, c_{44}$ | $p_{11}, p_{12}, p_{44}$ |
| Trigonal | 1 threefold rotation axis | LiNbO3 | $\epsilon_1, \epsilon_3$ | $c_{11}, c_{12}, c_{13}, c_{14}, c_{33}$ | $p_{11}, p_{12}, p_{13}, p_{14}, p_{31}, p_{33}, p_{41}, p_{44}$ |

The full form of the material tensors for each crystal class is as follows:

**Isotropic**

$$\epsilon_I = \begin{bmatrix} \epsilon_1 & 0 & 0 \\ 0 & \epsilon_1 & 0 \\ 0 & 0 & \epsilon_1 \end{bmatrix}$$

$$c_{IJ} = \begin{bmatrix} c_{11} & c_{12} & c_{12} & 0 & 0 & 0 \\ c_{12} & c_{11} & c_{12} & 0 & 0 & 0 \\ c_{12} & c_{12} & c_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & c_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{44} \end{bmatrix}$$

$$p_{IJ} = \begin{bmatrix} p_{11} & p_{12} & p_{12} & 0 & 0 & 0 \\ p_{12} & p_{11} & p_{12} & 0 & 0 & 0 \\ p_{12} & p_{12} & p_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & p_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & p_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & p_{44} \end{bmatrix}$$

where $c_{44} = (c_{11} - c_{12})/2$ and $p_{44} = (p_{11} - p_{12})/2$. These quantities are related to the *Lame parameters* $\mu = c_{44}$ and $\lambda = c_{11}$ which may in turn be expressed in terms of the Young's modulus $E$ and Poisson ratio $\nu$:

$$\mu = \frac{E}{2(1+\nu)}, \qquad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$$

**Cubic**

The matrix expressions are identical to the isotropic case except that

$c_{44}$ and $p_{44}$ are now independent quantities that must be specified directly.

**Trigonal**

$$\epsilon_I = \begin{bmatrix} \epsilon_1 & 0 & 0 \\ 0 & \epsilon_1 & 0 \\ 0 & 0 & \epsilon_3 \end{bmatrix}$$

$$c_{IJ} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & 0 & 0 \\ c_{12} & c_{11} & c_{13} & -c_{14} & 0 & 0 \\ c_{13} & c_{13} & c_{33} & 0 & 0 & 0 \\ c_{14} & -c_{14} & 0 & c_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{44} & c_{14} \\ 0 & 0 & 0 & 0 & c_{14} & (c_{11} - c_{12})/2 \end{bmatrix}$$

$$p_{IJ} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} & 0 & 0 \\ p_{12} & p_{11} & p_{13} & -p_{14} & 0 & 0 \\ p_{31} & p_{31} & p_{33} & 0 & 0 & 0 \\ p_{41} & -p_{41} & 0 & p_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & p_{44} & p_{41} \\ 0 & 0 & 0 & 0 & p_{14} & (p_{11} - p_{12})/2 \end{bmatrix}$$

WRITE ME

### 11.1.2 Required tensor components

WRITE ME

CHAPTER

TWELVE

# NUMERICAL FORMULATION OF THE FINITE ELEMENT PROBLEMS

## 12.1 Introduction

In this chapter we provide some detailed information on how the finite element problems defined in chapter *Background theory* are expressed as numerical problems that can be solved by standard matrix libraries.

We start with the elastic problem which is somewhat easier to understand since it is formulated with a simpler of elements.

## 12.2 Elastic modal problem

Recall from *Background theory* that the elastic wave equation has the form

$$\nabla \cdot \bar{T} + \Omega^2 \rho(x, y)\vec{u} = 0,$$

where we seek modal eigensolutions $(\vec{u}_n(\vec{r}_t)e^{iqz}, \Omega_n(q_z))$ for the elastic displacement vector field $\vec{U}(\vec{r})$:

$$\vec{U}(\vec{r}) = b\,\vec{u}_n(\vec{r}_t)e^{i(qz-\Omega_n(q)t)} + b^*\,\vec{u}_n^*(\vec{r}_t)e^{-i(qz-\Omega_n(q)t)},$$

where the position vectors are defined $\vec{r} \equiv (\vec{r}_t, z) \equiv (x, y, z)$. Refer to chapter *Background theory* for complete definitions of the density $\rho(\vec{r}_t)$, the rank-2 stress tensor $\bar{T}$, and its connection with the stiffness tensor $\mathbf{c}$ and strain tensor $\bar{S}$.

In the so-called *weak-formulation* of this eigenproblem, we seek pairs $(\vec{u}(\vec{r}_t)e^{iqz}, \Omega(q_z))$ so that for all "test" functions $\vec{v}(\vec{r})$, the following integral equation holds

$$\int_A \vec{v}^*(\vec{r}) \cdot \left(\nabla \cdot \bar{T} + \Omega^2 \rho \vec{u}(\vec{r})\right)\, \mathrm{d}A = 0.$$

### 12.2.1 Defining the FEM formulation

The weak form equation is a general mathematical statement that softens potential singularities in the original differential equation.

To develop a finite-element numerical algorithm of this statement, the functions $\vec{u}(\vec{r})$ and $\vec{v}(\vec{r})$ are expanded in a finite set of $N$ basis functions $g_m(\vec{r}_t)$:

$$\begin{aligned}\vec{u}(\vec{r}_t) &= \sum_{m=1}^{N} \begin{bmatrix} u_{m,x} \\ u_{m,y} \\ u_{m,z} \end{bmatrix} g_m(\vec{r}_t) \\ &= \sum_{m=1}^{N} \sum_{\sigma=x,y,z} u_{m,\sigma}\, g_m(\vec{r}_t)\mathbf{e}_\sigma\end{aligned}$$

for some set of coefficients $\vec{u}_h = (u_{1,x}, u_{1,y}, u_{1,z}, u_{2,x}, u_{2,y}, u_{2,z}, \ldots, u_{N,x}, u_{N,y}, u_{N,z})$, with separate coefficients for each Cartesian component of the field. Here, $\mathbf{e}_\sigma$ stands for the Cartesian unit vectors $\hat{x}, \hat{y}, \hat{z}$.

Note that the basis functions are scalar and each component of the displacement field is represented by a distinct degree of freedom. (This could be written in a vector notation defining $\vec{g}_{m,\sigma}(\vec{r}_t) = g_m(\vec{r}_t)\mathbf{e}_\sigma$, but there is no particular advantage in doing so.) Later, we see that this is handled differently in the electromagnetic problem where the basis functions are chosen to have a non-trivial vector character.

### FEM basis functions

In NumBAT, the $g_m(\vec{r}_t)$ are chosen as piecewise quadratic polynomials defined on each domain of an irregularly shaped triangular grid. These basis functions or *elements* are known as Lagrange P2 polynomials illustrated in the figure below.

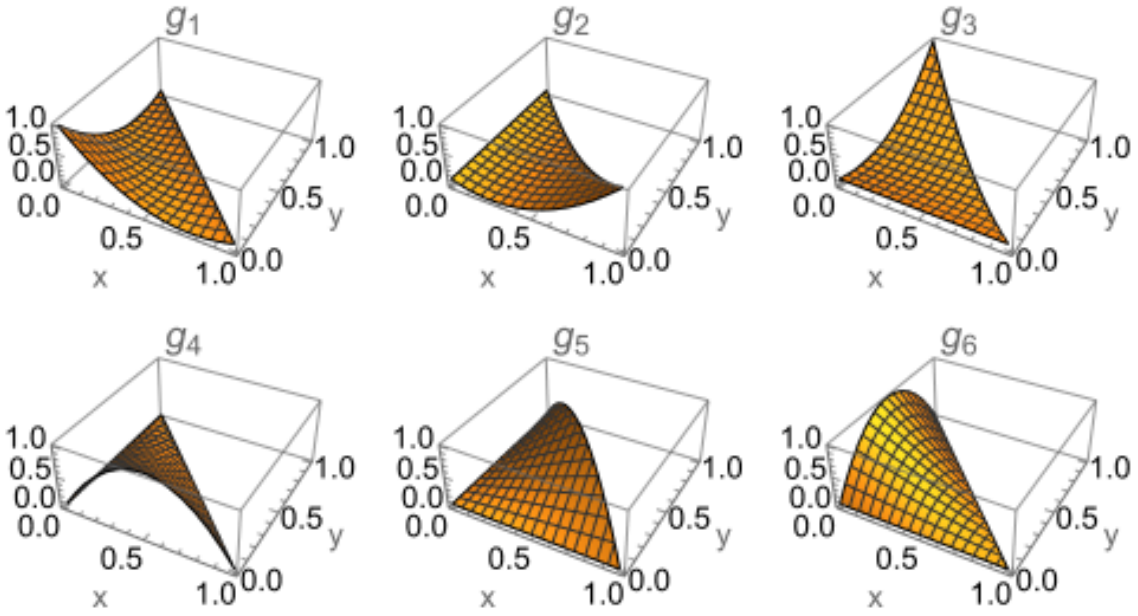



Fig. 1: Lagrange P2 polynomial scalar basis functions.

The standard polynomials are defined on the "unit" triangle with vertices or *nodes* at $\vec{r}_1 = (0,0)$, $\vec{r}_2 = (1,0)$, and $\vec{r}_3 = (0,1)$. Three additional nodes 4,5,6 label the mid-points at $\vec{r}_4 = (\frac{1}{2},0)$, $\vec{r}_5 = (0,\frac{1}{2})$, and $\vec{r}_6 = (\frac{1}{2},\frac{1}{2})$.

Observe that each polynomial takes the value zero at all nodes except the one corresponding to its label:

$$g_i(\vec{r}_j) = \delta_{ij},$$

and so frequently we can identify the nodes and basis functions of the same index.

The triangles in the FEM grid have arbitrary shapes and node locations, so that the values and overlap integrals of the functions are calculated for each triangle using an affine transformation.

### Determining the FEM matrices

The numerical task is to find eigenvector solutions for the vector of coefficients or *degrees of freedom* $\vec{u}_h$.

How many degrees of freedom are there? There are six P2 polynomials for each triangle in the grid. Since most nodes belong to more than one triangle and also due to the application of boundary conditions at the edge of the domain, the precise number of degrees of freedom depends on the exact arrangement of triangles in the FEM grid, but will be of order $n_{\text{dof}} = 10 n_t$ where $n_t$ is the number of triangles.

On inserting the field expansion into the weak form integral equation, the eigenproblem ultimately leads to the generalised linear eigenvalue equation (see [1]),

$$\mathrm{K}\vec{u}_h = \Omega^2 \mathrm{M}\vec{u}_h,$$

where the *stiffness* K and *mass* M operators capture the details of the waveguide structure. Note that the labels "stiffness" and "mass" are generic FEM terms, and are unrelated to the physical concepts of stiffness tensor and density in our specific elastic problem.

The values in the K and M operators are evaluated triangle by triangle. On *each* triangle, these operators are $18\times18$ matrices (6 nodes with 3 components). Labelling the elements by number $i = 1..6$ and component $\sigma = x, y, z$, we can write explicit formulae for each matrix element, with the rows associated with the test function $v$ and the columns with the coefficients of the eigenfunction $u_n$.

The mass operator comes from the second term in the weak-form wave equation:

$$\begin{aligned} M_{i,\sigma;j,\tau} &= \int_A \rho(\vec{r}_t) g_i(\vec{r}_t)\vec{e}_\sigma \cdot g_j(\vec{r}_t)\vec{e}_\tau \,\mathrm{d}x\mathrm{d}y \\ &= \delta_{\sigma,\tau} \int_A \rho(\vec{r}_t) g_i(\vec{r}_t) g_j(\vec{r}_t) \,\mathrm{d}x\mathrm{d}y \end{aligned}$$

The stiffness operator implements the negative of the first term in the weak-form wave equation (negative since it has been moved to the other side of the equation). It requires a bit more work to evaluate:

$$\begin{aligned} K[\vec{v},\vec{u}] &= -\int_A \vec{v}^*(\vec{r}_t)\cdot(\nabla\cdot\bar{T}[\vec{u}])\,\mathrm{d}A \\ &= -\int_A v_a^*(\nabla\cdot\bar{T}[\vec{u}])_a \,\mathrm{d}A \\ &= -\int_A v_a^*(\partial_b T_{ab})\,\mathrm{d}A \\ &= \int_A (\partial_b v_a^*) T_{ab}\,\mathrm{d}A \\ &= \int_A (\partial_b v_a^*) c_{abcd} S_{cd}\,\mathrm{d}A \\ &= \int_A (\partial_b v_a^*) c_{abcd} \tfrac{1}{2}(\partial_c u_d + \partial_d u_c)\,\mathrm{d}A \\ &= \frac{c_{abcd}}{2}\int_A (\partial_b v_a^*)(\partial_c u_d + \partial_d u_c)\,\mathrm{d}A \end{aligned}$$

Now to find $K_{i,\sigma;j,\tau}$ we can set $v_a \to g_i\delta_{a,\sigma}e^{iqz}$, $u_c \to g_j\delta_{c,\tau}e^{iqz}$ and $u_d \to g_j\delta_{d,\tau}e^{iqz}$, to obtain

$$\begin{aligned} K_{i,\sigma;j,\tau} &= \frac{c_{abcd}}{2}\int_A (\partial_b g_i\delta_{a,\sigma}e^{-iqz})(\partial_c g_j\delta_{d,\tau}e^{iqz} + \partial_d g_j\delta_{c,\tau}e^{iqz})\,\mathrm{d}A \\ &= \frac{c_{\sigma bcd}}{2}\int_A (\partial_b g_i e^{-iqz})(\partial_c g_j\delta_{d,\tau}e^{iqz} + \partial_d g_j\delta_{c,\tau}e^{iqz})\,\mathrm{d}A \\ &= \frac{c_{\sigma bc\tau}}{2}\int_A (\partial_b g_i e^{-iqz})(\partial_c g_j e^{iqz})\,\mathrm{d}x\mathrm{d}y + \frac{c_{\sigma b\tau d}}{2}\int_A (\partial_b g_i e^{-iqz})(\partial_d g_j e^{iqz})\,\mathrm{d}x\mathrm{d}y \end{aligned}$$

Writing the integral $\int_A f\mathrm{d}A = \langle f\rangle$ these terms can be evaluated as

$$\begin{aligned} G^{ij}_{bc} &= \langle(\partial_b g_i e^{-iqz})(\partial_c g_j e^{iqz})\rangle \\ &= \langle(-iqg_i\delta_{bz} + (\partial_b g_i)(1-\delta_{bz}))(iqg_j\delta_{cz} + (\partial_c g_j)(1-\delta_{cz}))\rangle \\ &= \langle q^2 g_i g_j \delta_{bz}\delta_{cz} - iqg_i(\partial_c g_j)\delta_{bz}(1-\delta_{cz}) \\ &\quad + iq(\partial_b g_i)g_j(1-\delta_{bz})\delta_{cz} + (\partial_b g_i)(\partial_c g_j)(1-\delta_{bz})(1-\delta_{cz})\rangle \\ &= \begin{bmatrix} \langle(\partial_x g_i)(\partial_x g_j)\rangle & \langle(\partial_x g_i)(\partial_y g_j)\rangle & \langle(\partial_x g_i)(iqg_j)\rangle \\ \langle(\partial_y g_i)(\partial_x g_j)\rangle & \langle(\partial_y g_i)(\partial_y g_j)\rangle & \langle(\partial_y g_i)(iqg_j)\rangle \\ \langle(-iqg_i)(\partial_x g_j)\rangle & \langle(-iqg_i)(\partial_y g_j)\rangle & q^2\langle g_i g_j\rangle \end{bmatrix}_{bc} \\ &= \begin{bmatrix} \langle g_{i;x} g_{j;x}\rangle & \langle g_{i;x} g_{j;y}\rangle & iq\langle g_{i;x} g_j\rangle \\ \langle g_{i;y} g_{j;x}\rangle & \langle g_{i;y} g_{j;y}\rangle & iq\langle g_{i;y} g_j\rangle \\ -iq\langle g_i g_{j;x}\rangle & -iq\langle g_i g_{j;y}\rangle & q^2\langle g_i g_j\rangle \end{bmatrix}_{bc} \end{aligned}$$

which at last gives

$$\begin{aligned} K_{i,\sigma;j,\tau} &= \frac{c_{\sigma bc\tau}}{2} G^{ij}_{bc} + \frac{c_{\sigma b\tau d}}{2} G^{ij}_{bd} \\ &= \frac{1}{2}\left( c_{\sigma bc\tau} G^{ij}_{bc} + c_{\sigma b\tau c} G^{ij}_{bc} \right) \\ &= \frac{1}{2}\left( c_{\sigma bc\tau} G^{ij}_{bc} + c_{\sigma bc\tau} G^{ij}_{bc} \right) \\ &= c_{\sigma bc\tau} G^{ij}_{bc} \end{aligned}$$

using the symmetries $c_{ijkl} = c_{jikl} = c_{ijlk}$.

Evaluating a couple of these elements using the Voigt notation for the stiffness tensor gives

$$\begin{aligned} K_{x,i;x,j} &= c_{xbcx} G^{ij}_{bc} \\ &= \big[ c_{xxxx} G^{ij}_{xx} + c_{xxyx} G^{ij}_{xy} + c_{xxzx} G^{ij}_{xz} + c_{xyxx} G^{ij}_{yx} + c_{xyyx} G^{ij}_{yy} \\ &\quad + c_{xyzx} G^{ij}_{yz} + c_{xzxx} G^{ij}_{zx} + c_{xzyx} G^{ij}_{zy} + c_{xzzx} G^{ij}_{zz} \big] \\ &= \big[ C_{11} G^{ij}_{xx} + C_{16} G^{ij}_{xy} + c_{15} G^{ij}_{xz} + c_{66} G^{ij}_{yx} + c_{66} G^{ij}_{yy} \\ &\quad + c_{65} G^{ij}_{yz} + c_{51} G^{ij}_{zx} + c_{56} G^{ij}_{zy} + c_{55} G^{ij}_{zz} \big] \\ K_{x,i;y,j} &= c_{xbcy} G^{ij}_{bc} \\ &= \big[ c_{xxxy} G^{ij}_{xx} + c_{xxyy} G^{ij}_{xy} + c_{xxzy} G^{ij}_{xz} + c_{xyxy} G^{ij}_{yx} + c_{xyyy} G^{ij}_{yy} \\ &\quad + c_{xyzy} G^{ij}_{yz} + c_{xzxy} G^{ij}_{zx} + c_{xzyy} G^{ij}_{zy} + c_{xzzy} G^{ij}_{zz} \big] \\ &= \big[ C_{16} G^{ij}_{xx} + C_{12} G^{ij}_{xy} + c_{14} G^{ij}_{xz} + c_{66} G^{ij}_{yx} + c_{62} G^{ij}_{yy} \\ &\quad + c_{64} G^{ij}_{yz} + c_{56} G^{ij}_{zx} + c_{52} G^{ij}_{zy} + c_{55} G^{ij}_{zz} \big] \end{aligned}$$

### 12.2.2 Solving the numerical problem

The entire K and M matrices have dimension $n_{\text{dof}} \times n_{\text{dof}}$ and are filled output by performing the above operation for each triangle in turn. But since only degrees of freedom that share a triangle can give nonzero terms, they are overwhelmingly sparse matrices. In NumBAT they are represented using the Compressed Sparse Column (CSC) format. Additional book-keeping is required in that most degrees of freedom are involved in multiple triangles but in each case must be mapped back to their unique row and column.

These matrices are constructed in the source files *build_fem_ops_ac.f90* which loops over all triangles and *make_elt_femops_ac.f90* which evaluates the operators for one triangle.

This matrix eigenproblem is then solved using standard linear algebra numerical libraries including ARPACK-NG, UMFPACK on top of the LAPACK and BLAS libraries. These libraries are installed separately by the user allowing selection of implementations that are optimal for the local operating system and hardware.

Once the coefficients $\vec{u}_h$ are known for a particular mode, they can be interpolated onto a rectangular grid for plotting. Note that evaluation of the key SBS coupling integrals is performed on the original FEM grid for accuracy.

## 12.3 Electromagnetic problem

We closely follow the exposition in [4].

Expressed in the modal form $\vec{E}(\vec{r}) = [\vec{E}_t, E_z] e^{i\beta z}$, the wave equation becomes the pair of equations:

$$\begin{aligned} \nabla_t \times \left( \frac{1}{\mu} (\nabla_t \times \vec{E}_t) \right) - \omega^2 \epsilon \vec{E}_t &= \beta^2 \frac{1}{\mu} (\nabla \hat{E}_z - \vec{E}_t) \\ -\nabla_t \cdot (\frac{1}{\mu} \vec{E}_t) + \nabla_t \cdot (\frac{1}{\mu} \nabla_t \hat{E}_z) + \omega^2 \epsilon \hat{E}_z &= 0, \end{aligned}$$

where for convenience we have introduced $\hat{E}_z E_z = -E_z/\beta$, and in practice the permeability is always $mu=mu_0$.

In the weak-form formulation, we seek pairs $(\vec{E}, \beta)$ so that for all test functions $\vec{F}(\vec{r})$, the following equations hold

$$\left(\frac{1}{\mu}(\nabla_t \times \vec{E}_t), \nabla_t \times \vec{F}_t\right) - \omega^2 \left(\epsilon \vec{E}_t, \vec{F}_t\right) = \beta^2 \left(\frac{1}{\mu}(\nabla \hat{E}_z - \vec{E}_t), \vec{F}_t\right)$$
$$\left(\frac{1}{\mu}\vec{E}_t, \nabla_t F_z\right) - \left(\frac{1}{\mu}\nabla_t \hat{E}_z, \nabla_t F_z\right) + \omega^2 \left(\epsilon \hat{E}_z, F_z\right) = 0,$$

where for arbitrary terms $\vec{A}, \vec{B}$ the inner product is defined

$$(\vec{A}, \vec{B}) = \int (\vec{B})^* \cdot (\vec{A}) \, \mathrm{d}A.$$

### 12.3.1 Defining the FEM formulation

The basis sets used to construct the FEM formulation are more involved than for the elastic problem, with the transverse and longitudinal components represented differently. We introduce transverse vector basis functions $\vec{\phi}_i(x, y)$ and scalar longitudinal functions $\psi_i(x, y)$ so that a general field has the form

$$\begin{aligned}
\vec{E} =& \vec{E}_t + \hat{z} E_z \\
=& \sum_{i=1}^{M} e_{t,i} \, \vec{\phi}_i(\vec{r}) + \hat{z} \sum_{i=1}^{N} \hat{e}_{z,i} \, \psi_i(\vec{r}) \\
=& [\vec{\phi}_1, \vec{\phi}_2, \ldots, \vec{\phi}_M, \psi_1, \psi_2, \ldots, \psi_N, ] \begin{bmatrix} e_{t,1} \\ e_{t,2} \\ \cdots \\ e_{t,M} \\ \hat{e}_{z,1} \\ \hat{e}_{z,2} \\ \cdots \\ \hat{e}_{z,N} \end{bmatrix} \\
=& [\vec{\boldsymbol{\phi}}; \hat{z}\boldsymbol{\psi}] \begin{bmatrix} \mathbf{e}_t \\ \hat{\mathbf{e}}_z \end{bmatrix}
\end{aligned}$$

#### The basis functions

The longitudinal component of the field is represented by 10 P3 Lagrange polynomials. These have nodes at the three vertices, six one-third points along each edge, and the *barycentre* of the triangle at $\vec{r}_{10} = (\frac{1}{3}, \frac{1}{3})$:

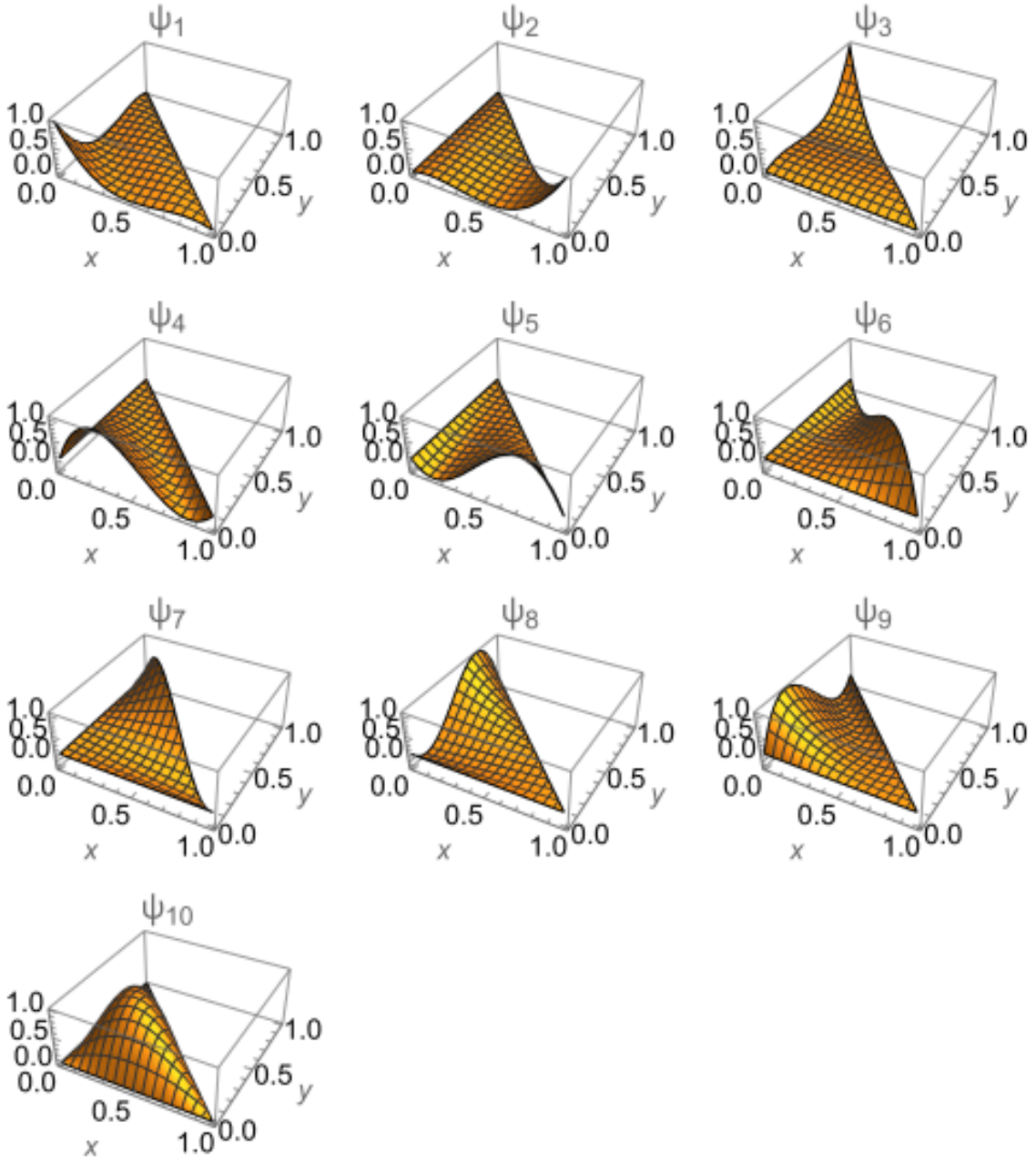



Fig. 2: Lagrange P3 polynomial scalar basis functions.

The transverse part of the field is represented by 12 basis functions composed from products of P2 polynomials and gradients of P1 polynomials. Three functions (shown in each row in the figure below) are associated with each of the barycentre and the three edge mid-points.

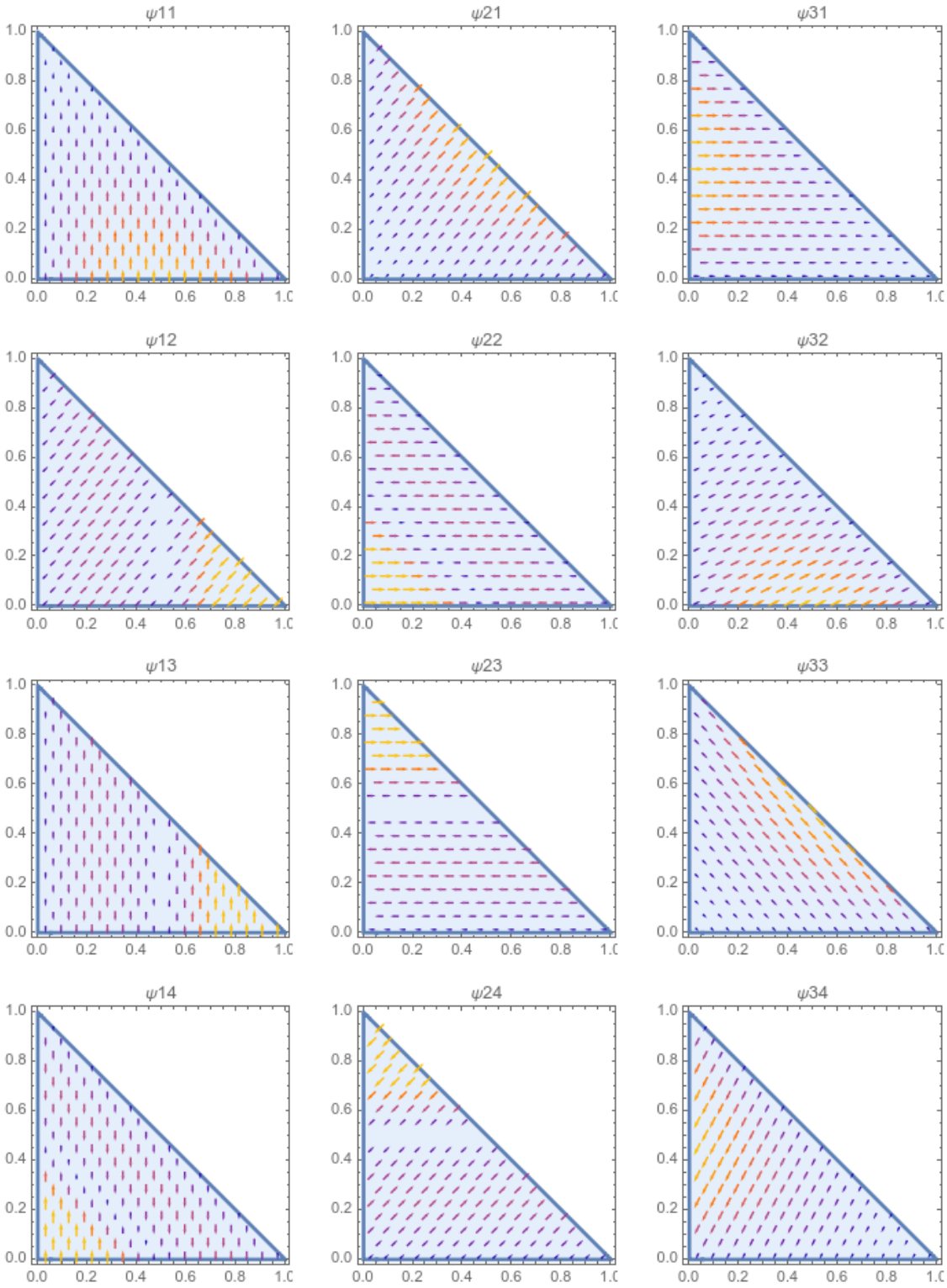



Fig. 3: Lagrange P3 polynomial scalar basis functions.

Hence in the basis function expansion above, we have $M = 12$ and $N = 10$.

### Determining the FEM matrices

An arbitrary inner product $(\mathcal{L}_1 E, \mathcal{L}_2 F)$ with $\mathcal{F} = a\vec{\phi}_m + b\psi_m \hat{z}$ where $a$ and $b$ are zero or one, expands to

$$\begin{aligned}
(\mathcal{L}_1\vec{E}, \mathcal{L}_2\vec{F}) =& \int (\mathcal{L}_2\vec{F})^* \cdot (\mathcal{L}_1\vec{E})\,\mathrm{d}A \\
=& \int (\mathcal{L}_2 a\vec{\phi}_m + b\psi_m\hat{z})^* \cdot (\mathcal{L}_1[\vec{\boldsymbol{\phi}}; \hat{z}\boldsymbol{\psi}] \begin{bmatrix}\mathbf{e}_t \\ \hat{\mathbf{e}}_z\end{bmatrix}) \\
=& a\int (\mathcal{L}_2\vec{\phi}_m)^* \cdot (\mathcal{L}_1\vec{\phi}_n)\,\mathrm{d}A\, e_{t,n} + b\int (\mathcal{L}_2\psi_m\hat{z})^* \cdot (\mathcal{L}_1\vec{\phi}_n)\,\mathrm{d}A\, e_{t,n} \\
&+ a\int (\mathcal{L}_2\vec{\phi}_m)^* \cdot (\mathcal{L}_1\psi_n\hat{z})\,\mathrm{d}A\, \hat{e}_{z,n} + b\int (\mathcal{L}_2\psi_m\hat{z})^* \cdot (\mathcal{L}_1\psi_n\hat{z})\,\mathrm{d}A\, \hat{e}_{z,n} \\
=& \mathcal{L}_{tt}\mathbf{e}_t + \mathcal{L}_{zt}\mathbf{e}_z + \mathcal{L}_{tz}\hat{\mathbf{e}}_z + \mathcal{L}_{zz}\hat{\mathbf{e}}_z.
\end{aligned}$$

With these definitions we can identify the matrices

$$\begin{aligned}
K_{mn} =& \int \left[(\nabla_t \times \vec{\phi}_m)^* \cdot \frac{1}{\mu}(\nabla_t \times \vec{\phi}_n) - \omega^2 \vec{\phi}_m^* \cdot (\epsilon\vec{\phi}_n)\right] \mathrm{d}A \\
K_{zt} =& \int (\nabla_t\psi_m)^* \cdot \frac{1}{\mu}\vec{\phi}_n\,\mathrm{d}A \\
K_{zz} =& \omega^2 \int \psi_m^* \cdot \epsilon\psi_n\,\mathrm{d}A \\
M_{tt} =& -\int \vec{\phi}_m^* \cdot \vec{\phi}_n\,\mathrm{d}A
\end{aligned}$$

Then the eigenproblem to be solved is the generalised linear problem

$$\begin{bmatrix}[K_{tt}] & [0] \\ [K_{zt}] & [K_{zz}]\end{bmatrix} \begin{bmatrix}e_{t,h} \\ e_{z,h}\end{bmatrix} = \beta^2 \begin{bmatrix}[M_{tt}] & [K_{zt}]^T \\ [0] & [0]\end{bmatrix} \begin{bmatrix}e_{t,h} \\ e_{z,h}\end{bmatrix}.$$

By swapping the sides of the second row, the two matrices involved become symmetric:

$$\begin{bmatrix}[K_{tt}] & [0] \\ [0] & [0]\end{bmatrix} \begin{bmatrix}e_{t,h} \\ e_{z,h}\end{bmatrix} = \beta^2 \begin{bmatrix}[M_{tt}] & [K_{zt}]^T \\ [K_{zt}] & [K_{zz}]\end{bmatrix} \begin{bmatrix}e_{t,h} \\ e_{z,h}\end{bmatrix},$$

which is ideally posed to solve using standard numerical libraries.

CHAPTER

# THIRTEEN

# PYTHON INTERFACE API

This chapter provides an auto-generated summary of the NumBAT Python API.

The API consists of several core modules:

- `numbat`, for creating the top-level NumBAT application;
- `materials`, for defining waveguide materials and their properties;
- `voigt`, for manipulating tensor quantities.
- `structure`, for constructing waveguides from materials;
- `modecalcs`, for the core calculation of electromagnetic and acoustic modes;
- `integration`, for performing calculations relating to SBS gain;
- `plotting`, for creating output plots of modes and gain functions.

For the most part, users should only encounter the `numat` and `integration` modules.

## 13.1 numbat module

The `numbat` module contains the `NumBATApp` through which the bulk of the NumBAT API is accessed.

Creating a `NumBATApp` object is normally the first main step in a NumBAT script. NumBAT main application module.

This module provides the main application singleton, handles environment setup, and provides interfaces for running and managing NumBAT simulations, including output management, version checking, and integration with the Fortran backend.

Key classes and functions:

- *NumBATApp* – Singleton class for managing global NumBAT state and configuration.
- *NumBATPlotPrefs()* – Returns the global plot preferences object.
- *load_simulation()* – Loads a saved simulation from disk.
- *version()* – Returns the NumBAT version string.

**class** numbat.**NumBATApp**(**args*, ***kwargs*)

Bases: `object`

Singleton class for managing the NumBAT application state, configuration, and output paths. Provides methods for output management, environment setup, and version checking. Use NumBATApp() to get the singleton instance.

**can_multiprocess**()

Return True if the platform supports multiprocessing (Linux only).

final_report(*outprefix=''*)

Return a string summarizing the simulation run and any warnings. Optionally writes warnings to a file.

**Parameters**
**outprefix** (*str*) – Output prefix for warnings file (optional).

**Returns**
Summary report string.

**Return type**
str

is_linux()

Return True if running on Linux.

is_macos()

Return True if running on macOS.

is_windows()

Return True if running on Windows.

make_structure(**args*, ***kwargs*)

Create and return a Structure object for the simulation.

outdir()

Return the current global output directory as a string.

outdir_fields_path(*prefix=''*)

Return the output directory path for modal fields, creating it if necessary.

**Parameters**
**prefix** (*str*) – Output prefix (optional).

**Returns**
Path object for the fields directory.

**Return type**
Path

outpath(*prefix=''*)

Return the full output path for a given prefix.

**Parameters**
**prefix** (*str*) – Output prefix (optional).

**Returns**
Full output path.

**Return type**
str

outprefix()

Return the current global output prefix as a string.

path_gmsh()

Return the path to the gmsh executable.

path_mesh_templates()

Return the path to the mesh templates directory.

plotfile_ext()

Return the current plot file extension as a string.

**set_outdir**(*outdir*)

Set the current global output directory, creating it if necessary.

**Parameters**
**outdir** (*str*) – Output directory path.

**set_outprefix**(*s*)

Set the global output file prefix.

**Parameters**
**s** (*str*) – Output prefix.

**version**()

Return the NumBAT version string.

numbat.**NumBATPlotPrefs**()

Returns the global NumBAT plot preferences object.

numbat.**load_simulation**(*prefix*)

Loads a saved Simulation object from disk using the given prefix.

**Parameters**
**prefix** (*str*) – Prefix for the simulation file.

**Returns**
The loaded simulation object.

**Return type**
*Simulation*

numbat.**version**()

Returns the NumBAT version string.

## 13.2 materials module

The `materials` module provides functions for specifying all relevant optical and elastic properties of waveguide materials.

The primary class is *materials.Material* however users will rarely use this class directly. Instead, we generally specify material properties by writing new `.json` files stored in the folder `backend/materials_data`. Materials are then loaded to build waveguide structures using the function *materials.make_material()*.

**exception** materials.**BadMaterialFileError**

Bases: `Exception`

Exception raised for errors in material file parsing or validation.

**class** materials.**Material**(*json_data*, *filename*)

Bases: `object`

Represents a waveguide material, including optical, elastic, and piezoelectric properties. Should be constructed via materials.get_material().

**Vac_Rayleigh**()

For an isotropic material, return the Rayleigh wave elastic phase velocity.

**Vac_longitudinal**()

For an isotropic material, return the longitudinal (P-wave) elastic phase velocity.

**Vac_phase**()

Return the three acoustic phase velocities for propagation along z for the current orientation.

**Vac_shear**()

For an isotropic material, return the shear (S-wave) elastic phase velocity.

**construct_crystal**()

Construct the tensors for the crystal based on its symmetry class.

**construct_crystal_cubic**()

Return lists of independent and dependent elements for cubic crystals.

**construct_crystal_general**()

Fill all tensor elements for a general anisotropic crystal from the parameter dictionary.

**construct_crystal_hexagonal**()

Return lists of independent and dependent elements for hexagonal crystals.

**construct_crystal_isotropic**()

Construct the tensors for an isotropic crystal from the parameter dictionary.

**construct_crystal_trigonal**()

Return lists of independent and dependent elements for trigonal crystals.

**copy**()

Return a deep copy of the material object.

**disable_piezoelectric_effects**()

Disable piezoelectric effects for this material (if supported).

**elastic_properties**(*chop=False*, *chop_rtol=1e-10*, *chop_atol=1e-12*)

Return a string describing the elastic properties of the material.

**em_phase_index**()

Return the three optical phase indices for propagation along z for the current orientation.

**enable_piezoelectric_effects**()

Enable piezoelectric effects for this material (if supported).

**full_str**(*chop=False*, *chop_rtol=1e-10*, *chop_atol=1e-12*)

Return a detailed string representation of the material, including all tensors and piezo properties.

**get_stiffness_for_kappa**(*vkap*)

Get the stiffness tensor, piezo-hardened if piezoelectric effects are active.

**Parameters**
**vkap** (*np.ndarray*) – Propagation direction (3-vector).

**Returns**
Stiffened stiffness tensor.

**Return type**
*VoigtTensor4_IJ*

**has_elastic_properties**()

Returns True if the material has elastic properties defined.

**is_isotropic**()

Returns True if the material is isotropic.

**is_vacuum**()

Returns True if the material is vacuum.

**make_crystal_axes_plot**(*pref*)

Build a crystal coordinates diagram using an external application. Returns the output PNG filename.

**optical_properties**()

Return a string describing the optical properties of the material.

**piezo_supported**()

Returns True if the material supports piezoelectric effects.

**plot_bulk_dispersion_3D**(*pref*)

Generate isocontour surfaces of the bulk dispersion in 3D k-space. Returns the filename of the generated image.

**plot_bulk_dispersion_all**(*pref*, *label=None*, *show_poln=True*, *flip_x=False*, *flip_y=False*, *cut_plane='xz'*, *mark_velocities=()*)

Plot the bulk dispersion and ray surfaces in the x-z plane for the current orientation (all modes). Returns the filename of the generated image.

Solving the Christoffel equation: D C D^T u = - ho v_p^2 u, for eigenvalue v_p and eigengector u.

C is the Voigt form stiffness. D = [ [kapx 0 0 0 kapz kapy ] [0 kapy 0 kapz 0 kapx ] [0 0 kapz kapy kapx 0]] where kap=(cos phi, 0, sin phi).

Returns filename of the generated image.

**plot_bulk_dispersion_ivp**(*pref*, *label=None*, *show_poln=True*, *flip_x=False*, *flip_y=False*, *cut_plane='xz'*, *mark_velocities=()*)

Plot the bulk dispersion as inverse phase velocity. Returns the filename of the generated image.

**plot_bulk_dispersion_vg**(*pref*, *label=None*, *show_poln=True*, *flip_x=False*, *flip_y=False*, *cut_plane='xz'*, *mark_velocities=()*)

Plot the bulk dispersion as the group velocity (ray surface). Returns the filename of the generated image.

**plot_photoelastic_IJ**(*prefix*, *v_comps*)

Plot photoelastic tensor components as a function of rotation angle about the y-axis.

**Parameters**

- **prefix** (*str*) – Prefix for output file names.
- **v_comps** (*list*) – List of element strings (e.g., '11', '12').

**reset_orientation**()

Restore the material and all tensors to their original orientation.

**rotate**(*rot_axis_spec*, *theta*, *save_rotated_tensors=False*)

Rotate the crystal axes and all tensors by theta radians about the specified axis.

**Parameters**

- **rot_axis_spec** (*str, tuple, list, or np.ndarray*) – Axis specification.
- **theta** (*float*) – Angle in radians.
- **save_rotated_tensors** (*bool*) – If True, save rotated tensors to CSV (default False).

**Returns**
The rotated material object.

**Return type**
*Material*

**rotate_axis**(*rotation_axis*, *theta*, *save_rotated_tensors=False*)

Deprecated. Use rotate().

**set_crystal_axes**(*va*, *vb*, *vc*)

Set the crystal axes vectors.

**Parameters**

- **va** (*np.ndarray*) – Crystal axis vectors.

- **vb** (*np.ndarray*) – Crystal axis vectors.
- **vc** (*np.ndarray*) – Crystal axis vectors.

**set_orientation**(*label*)

Set the orientation of the crystal to a specific named orientation (e.g., 'x-cut', '111').

**Parameters**
**label** (*str*) – Orientation label defined in the material file.

**set_refractive_index**(*nr*, *ni=0.0*)

Set the complex refractive index for the material.

**Parameters**

- **nr** (*float*) – Real part.
- **ni** (*float*) – Imaginary part (default 0.0).

**class** materials.**MaterialLibrary**

Bases: object

Manages the loading and retrieval of material definitions from JSON files in the material_data directory.

**get_material**(*matname*)

Retrieve a Material object by name.

**Parameters**
**matname** (*str*) – Name of the material to retrieve.

**Returns**
The requested material object.

**Return type**
*Material*

**Raises**
**SystemExit** – If the material is not found.

**class** materials.**PiezoElectricProperties**(*d_piezo*, *stiffness_cE_IJ*)

Bases: object

Encapsulates piezoelectric tensor properties and their manipulation for a material. Handles loading, rotation, and conversion between different piezoelectric tensor forms.

**d_iJ_to_d_ijk**()

Convert the d_iJ tensor to full d_ijk notation (not implemented).

**d_ijk_to_d_iJ**()

Convert the d_ijk tensor to Voigt d_iJ notation (not implemented).

**make_piezo_stiffened_stiffness_for_kappa**(*v_kap*)

Returns the piezo-stiffened stiffness tensor for propagation along a given direction. See Auld 8.147.

**Parameters**
**v_kap** (*np.ndarray*) – Propagation direction (3-vector).

**Returns**
Piezo-stiffened stiffness tensor.

**Return type**
*VoigtTensor4_IJ*

**reset_orientation**()

Restore all piezoelectric tensors to their original (unrotated) state.

**rotate**(*matR*)

Rotate all piezoelectric tensors by a given SO(3) rotation matrix.

**Parameters**
**matR** (*np.ndarray*) – 3x3 rotation matrix.

materials.**compare_bulk_dispersion**(*mat1*, *mat2*, *pref*)

materials.**disprel_rayleigh**(*vR*, *vl*)

materials.**find_Rayleigh_velocity**(*Vs*, *Vl*)

Find Rayleigh velocity v_q for isotropic mode with velocities Vs and Vl.

materials.**isotropic_stiffness**(*E*, *v*)

Calculate the stiffness matrix components of isotropic materials, given the two free parameters.

Ref: www.efunda.com/formulae/solid_mechanics/mat_mechanics/hooke_isotropic.cfm

**Parameters**

- **E** (*float*) – Youngs modulus
- **v** (*float*) – Poisson ratio

materials.**make_material**(*s*)

## 13.3 voigt module

The voigt module provides functions for performing a number of tensor operations with regular and Voigt style tensors.

class voigt.**PlainTensor2_ij**(*param_nm*, *physical_name=''*, *physical_symbol=''*, *unit=None*, *is_complex=False*, *is_symmetric=False*)

Bases: object

A class for representing rank 2 tensors with ij indexing in [0..2, 0..2] (regular indexing in the first slot, Voigt in the last two slots)

**as_str**(*chop=False*, *rtol=1e-10*, *atol=1e-12*)

Return a string representation of the tensor, formatted for display.

**Parameters**

- **chop** (*bool*) – Whether to chop small values to zero.
- **rtol** (*float*) – Relative tolerance for chopping.
- **atol** (*float*) – Absolute tolerance for chopping.

**Returns**
Formatted string representation of the tensor.

**Return type**
str

**copy**()

Return a deep copy of the tensor object.

**fill_random**()

Fill the tensor with random values (uniform in [0,1)).

**rotate**(*mat_R*)

Rotate the tensor using the SO(3) matrix mat_R.

**Parameters**
**mat_R** (*np.ndarray*) – 3x3 rotation matrix.

**set_from_00matrix**(*mat00*)

Set the tensor values from a 3x3 matrix.

**Parameters**
**mat00** (*np.ndarray*) – 3x3 matrix.

**set_from_params**(*d_params*)

Set tensor elements from a parameter dictionary, handling symmetry and complex values.

**Parameters**
**d_params** (*dict*) – Dictionary of parameter values.

**value**()

Return a copy of the tensor's matrix data.

voigt.**Voigt3_iJ_to_ijk**(*mat_iJ*, *fac2mul=False*)

Convert a 3x6 Voigt-indexed tensor to a 3x3x3 tensor, optionally multiplying off-diagonal elements by 2.

**Parameters**

- **mat_iJ** (*np.ndarray*) – 3x6 Voigt-indexed tensor.
- **fac2mul** (*bool*) – Whether to multiply off-diagonal elements by 2.

**Returns**
3x3x3 tensor.

**Return type**
np.ndarray

voigt.**Voigt3_ijk_to_iJ**(*mat_ijk*, *fac2mul=False*)

Convert a 3x3x3 tensor to a 3x6 Voigt-indexed tensor, optionally dividing off-diagonal elements by 2.

**Parameters**

- **mat_ijk** (*np.ndarray*) – 3x3x3 tensor.
- **fac2mul** (*bool*) – Whether to divide off-diagonal elements by 2.

**Returns**
3x6 Voigt-indexed tensor.

**Return type**
np.ndarray

**class** voigt.**VoigtTensor3_iJ**(*param_nm*, *physical_name=''*, *physical_symbol=''*, *unit=None*, *transforms_with_factor_2=False*)

Bases: object

A class for representing rank 3 tensors with iJ indexing (regular indexing in the first slot, Voigt in the last two slots)

**as_str**(*chop=False*, *rtol=1e-10*, *atol=1e-12*)

Return a string representation of the tensor, formatted for display.

**Parameters**

- **chop** (*bool*) – Whether to chop small values to zero.
- **rtol** (*float*) – Relative tolerance for chopping.
- **atol** (*float*) – Absolute tolerance for chopping.

**Returns**
Formatted string representation of the tensor.

**Return type**
str

**as_transpose_iJ**()

Return the transpose of the tensor (shape [6,3]).

**copy**()

Return a deep copy of the tensor object.

**elt_iJ**(*s_iJ*)

Return the value of a single element (by string index).

**Parameters**
**s_iJ** (*str*) – String index (e.g., 'x1').

**Returns**
Value of the element.

**Return type**
float

**fill_random**()

Fill the tensor with random values (uniform in [0,1)).

**refvalue**()

Return a reference to the tensor's matrix data (shape [3,6]).

**rotate**(*mat_R*)

Rotate the tensor using the SO(3) matrix mat_R, returning a new tensor.

**Parameters**
**mat_R** (*np.ndarray*) – 3x3 rotation matrix.

**set_elt_from_param_dict**(*s_iJ*, *d_params*)

Load a single element (by string index) from d_params.

**Parameters**

- **s_iJ** (*str*) – String index (e.g., 'x1').
- **d_params** (*dict*) – Dictionary of parameter values.

**set_elt_iJ**(*s_iJ*, *val*)

Set the value of a single element (by string index).

**Parameters**

- **s_iJ** (*str*) – String index (e.g., 'x1').
- **val** (*float*) – Value to set.

**set_from_00matrix**(*mat00*)

Set the tensor values from a 3x6 matrix.

**Parameters**
**mat00** (*np.ndarray*) – 3x6 matrix.

**set_from_params**(*d_params*)

Set tensor elements from a parameter dictionary.

**Parameters**
**d_params** (*dict*) – Dictionary of parameter values.

**set_from_structure_elts**(*d_params*, *elts_indep*, *elts_dep*, *depwhich*)

Set elements from known sets of independent and dependent values, with optional multipliers.

**Parameters**

- **d_params** (*dict*) – Dictionary of parameter values.
- **elts_indep** (*list*) – List of independent element indices.

- **elts_dep** (*dict*) – Dictionary of dependent element indices and their relations.
- **depwhich** (*int*) – Index of the multiplier to use.

**value**()

Return a copy of the tensor's matrix data (shape [3,6]).

class voigt.**VoigtTensor4_IJ**(*param_nm*, *physical_name=''*, *physical_symbol=''*, *unit=None*)

Bases: object

A class for representing rank 4 tensors in the compact Voigt representation.

Internally uses 1-based indexing like the notation. Externally it passes zero-based values.

**as_str**(*chop=False*, *rtol=1e-10*, *atol=1e-12*)

Return a string representation of the tensor, formatted for display.

**Parameters**

- **chop** (*bool*) – Whether to chop small values to zero.
- **rtol** (*float*) – Relative tolerance for chopping.
- **atol** (*float*) – Absolute tolerance for chopping.

**Returns**
Formatted string representation of the tensor.

**Return type**
str

**check_symmetries**()

Check that the tensor matrix is symmetric and positive definite.

**copy**()

Return a deep copy of the tensor object.

**dump_rawdata**()

Print the raw data of the tensor for debugging purposes.

**elt_s_IJ**(*s_IJ*)

Get an element by string index (e.g., '12', '23').

**Parameters**
**s_IJ** (*str*) – String index (e.g., '12').

**Returns**
Value at the specified indices.

**Return type**
float

**elt_specified**(*d_params*, *m*, *n*)

Check if a tensor element is specified in the parameter dictionary.

**fill_random**()

Fill the tensor with random values (uniform in [0,1)).

**load_isotropic_from_json**(*d_params*)

Load isotropic tensor values from the parameter dictionary.

**make_isotropic_tensor**(*m11*, *m12*, *m44*)

Build Voigt matrix from 3 parameters for isotropic geometry. (Actually, only two are independent.)

**Parameters**

- **m11** (*float*) – Diagonal value.
- **m12** (*float*) – Off-diagonal value.

- **m44** (*float*) – Shear value.

**refvalue**()

Return a reference to the tensor's matrix data (shape [6,6]).

**rotate**(*mat_R*)

Rotate the tensor using the SO(3) matrix mat_R and return a new tensor.

**Parameters**
**mat_R** (*np.ndarray*) – 3x3 rotation matrix.

**sIJ_to_rc**(*s_IJ*)

Convert a string index (e.g., '12') to row and column indices.

**Parameters**
**s_IJ** (*str*) – String index (e.g., '12').

**Returns**
(row, column) indices.

**Return type**
tuple

**set_elt_from_param_dict**(*I*, *J*, *d_params*)

Load a single element (by row and column indices) from d_params.

**Parameters**

- **I** (*int*) – Row index (1-based).
- **J** (*int*) – Column index (1-based).
- **d_params** (*dict*) – Dictionary of parameter values.

**set_elt_from_param_dict_as_str**(*s_IJ*, *d_params*)

Load a single element (by string index) from d_params.

**Parameters**

- **s_IJ** (*str*) – String index (e.g., '12').
- **d_params** (*dict*) – Dictionary of parameter values.

**set_elt_s_IJ**(*s_IJ*, *val*)

Set an element by string index (e.g., '12', '23').

**Parameters**

- **s_IJ** (*str*) – String index (e.g., '12').
- **val** (*float*) – Value to set.

**set_from_00matrix**(*mat00*)

Set the tensor values from a 6x6 matrix.

**Parameters**
**mat00** (*np.ndarray*) – 6x6 matrix.

**set_from_structure_elts**(*d_params*, *elts_indep*, *elts_dep*)

**value**()

Return a copy of the tensor's matrix data (shape [6,6]).

voigt.**from_Voigt = [None, (0, 0), (1, 1), (2, 2), (1, 2), (0, 2), (0, 1)]**

List mapping Voigt indices [1..6] to (i, j) pairs in [0..2]x[0..2].

voigt.**generate_voigt_rotation_matrix**(*matR*)

Generates the 6x6 transformation matrix M for rotating a rank 4 tensor expressed in 6x6 Voigt notation, given a 3x3 rotation matrix R.

The transformation is C_rotated = M @ C_unrotated @ M.T

**Parameters**
**matR** (*numpy.ndarray*) – A 3x3 NumPy array representing the rotation matrix R.

**Returns**

**A 6x6 NumPy array representing the transformation matrix M.**
Returns None if matR is not a 3x3 matrix.

**Return type**
numpy.ndarray

voigt.**kvec_to_symmetric_gradient**(*kvec*)

Calculate the 6x3 symmetric gradient operator for a plane wave with wavevector kvec.

**Parameters**
**kvec** (*array-like*) – Wavevector (kx, ky, kz).

**Returns**
6x3 symmetric gradient operator matrix.

**Return type**
np.ndarray

**Reference:**
Auld I, eq 1.53

voigt.**make_rotation_matrix**(*rot_axis_spec*, *theta*)

Return the SO(3) matrix corresponding to a rotation of theta radians about the specified axis.

**Parameters**

- **rot_axis_spec** (*str, tuple, list, or np.ndarray*) – Axis specification.
- **theta** (*float*) – Angle in radians.

**Returns**
3x3 rotation matrix.

**Return type**
np.ndarray

voigt.**parse_rotation_axis**(*rot_axis_spec*)

Convert a rotation axis specification to a standard unit 3-vector.

**Parameters**
**rot_axis_spec** (*str, tuple, list, or np.ndarray*) – Axis specification (e.g., 'x', 'y', 'z', or a 3-vector).

**Returns**
Normalized 3-element unit vector.

**Return type**
np.ndarray

voigt.**rotate_3vector**(*vec3*, *mat_R*)

Rotate a 3-vector using a given rotation matrix.

**Parameters**

- **vec3** (*np.ndarray*) – 3-element vector.
- **mat_R** (*np.ndarray*) – 3x3 rotation matrix.

**Returns**
Rotated 3-element vector.

**Return type**
np.ndarray

voigt.**strain_3mat_to_6col**(*Sm*)

Convert a 3x3 symmetric strain matrix to a 6-element Voigt strain vector.

**Parameters**
**Sm** (*np.ndarray*) – 3x3 symmetric strain matrix.

**Returns**
6-element strain vector in Voigt notation.

**Return type**
np.ndarray

voigt.**strain_6col_to_3mat**(*Sv*)

Convert a 6-element Voigt strain vector to a 3x3 symmetric strain matrix.

**Parameters**
**Sv** (*array-like*) – 6-element strain vector in Voigt notation.

**Returns**
3x3 symmetric strain matrix.

**Return type**
np.ndarray

voigt.**stress_3mat_to_6col**(*Tm*)

Convert a 3x3 symmetric stress matrix to a 6-element Voigt stress vector.

**Parameters**
**Tm** (*np.ndarray*) – 3x3 symmetric stress matrix.

**Returns**
6-element stress vector in Voigt notation.

**Return type**
np.ndarray

voigt.**stress_6col_to_3mat**(*Tv*)

Convert a 6-element Voigt stress vector to a 3x3 symmetric stress matrix.

**Parameters**
**Tv** (*array-like*) – 6-element stress vector in Voigt notation.

**Returns**
3x3 symmetric stress matrix.

**Return type**
np.ndarray

voigt.**to_Voigt3_index = array([[1, 6, 5], [6, 2, 4], [5, 4, 3]])**

Array mapping (i, j) indices in [0..2]x[0..2] to Voigt notation indices [1..6].

voigt.**to_Voigt_zerobase = array([[0, 5, 4], [5, 1, 3], [4, 3, 2]])**

Array mapping (i, j) indices in [0..2]x[0..2] to zero-based Voigt indices [0..5].

## 13.4 modes module

The modes module defines the main classes for displaying and interrogating optical and elastic modes.

class modes.**Mode**(*simres*, *m*)

Bases: object

This is a base class for both EM and AC modes.

**add_mode_data**(*d*)

Adds a dictionary of user-defined information about a mode.

**Parameters**
**d** (*dict*) – Dict of (str, data) tuples of user-defined information about a mode.

**analyse_mode**(*n_pts=501*, *ft=FieldType.EM_E*)

Perform a series of measurements on the mode *f* to determine polarisation fractions, second moment widths etc.

**Parameters**

- **v_x** (*array*) – Vector of x points.
- **v_y** (*array*) – Vector of y points.
- **m_Refx** (*array*) – Matrix of real part of fx.
- **m_Refy** (*array*) – Matrix of real part of fy.
- **m_Refz** (*array*) – Matrix of real part of fz.
- **m_Imfx** (*array*) – Matrix of imaginary part of fx.
- **m_Imfy** (*array*) – Matrix of imaginary part of fy.
- **m_Imfz** (*array*) – Matrix of imaginary part of fz.

**center_of_mass**()

Returns the centre of mass of the mode relative to the specified origin.

**Return type**
float

**center_of_mass_x**()

Returns the $x$ component moment of the centre of mass of the mode.

**Return type**
float

**center_of_mass_y**()

Returns the $y$ component moment of the centre of mass of the mode.

**Return type**
float

**clear_mode_plot_data**()

**field_fracs**()

Returns tuple (*fx*, *fy*, *fz*, *ft*) of "fraction" of mode contained in *x*, *y*, *z* or *t* (sum of transverse *x*+*y*) components.

Note that *fraction* is defined through a simple overlap integral. It does not necessarily represent the fraction of energy density in the component.

**Returns**
Tuple of mode fractions

**Return type**
tuple(float, float, float, float)

**get_mode_data**()

Return dictionary of user-defined information about the mode.

**Returns**
Dictionary of user-defined information about the mode.

**Return type**
dict(str, obj)

**is_AC**()

Returns true if the mode is an acoustic mode.

**Return type**
bool

**is_EM**()

Returns true if the mode is an electromagnetic mode.

**Return type**
bool

**is_poln_ex**()

Returns true if mode is predominantly x-polarised (ie if $fx$>0.7).

**Return type**
bool

**is_poln_ey**()

Returns true if mode is predominantly y-polarised (ie if $fy$>0.7).

**Return type**
bool

**is_poln_indeterminate**()

Returns true if transverse polarisation is neither predominantly $x$ or $y$ oriented.

**Return type**
bool

**plot_mode**(*comps=()*, *field_type=FieldType.EM_E*, *ax=None*, *n_pts=501*, *decorator=None*, *prefix=''*, *plot_params=<plotmodes.PlotParams2D object>*)

**plot_mode_1D**(*s_cut*, *val1*, *val2=None*, *comps=()*, *field_type=FieldType.EM_E*, *n_pts=501*, *prefix=''*, *plot_params=<plotmodes.PlotParams2D object>*)

Make a 1D plot of the field components of this mode along a line.

s_cut is a string determining the cut direction, with allowed values: 'x', 'y', or 'line'.

For s_cut='x', the function is plotted along y at fixed x=val1, where val1 is a float.

For s_cut='y', the function is plotted along x at fixed x=val1, where val1 is a float.

For s_cut='line', the function is plotted along a straight line from val1 to val2, where the latter are float tuples (x0,y0) and (x1,y1).

**plot_mode_H**(*comps*, *ax=None*, *n_pts=501*, *decorator=None*)

Plot magnetic field for EM modes.

**plot_mode_raw_fem**(*comps*)

Plot the requested field components on the sim mesh with no interpolation.

**plot_strain**()

**second_moment_widths**()

Returns the second moment widths $(w_x, w_y, \sqrt{w_x^2 + w_y^2})$ of the mode relative to the specified origin.

**Return type**
(float, float, float)

**set_r0_offset**(*x0*, *y0*)

Sets the transverse position in the grid that is to be regarded as the origin for calculations of center-of-mass.

This can be useful in aligning the FEM coordinate grid with a physically sensible place in the waveguide.

**Parameters**

- **x0** (*float*) – *x* position of nominal origin.
- **y0** (*float*) – *y* position of nominal origin.

**set_width_r0_reference**(*x0*, *y0*)

Set reference point for calculation of second moment width.

Positions are measured in microns.

**w0**()

Returns the combined second moment width $\sqrt{w_x^2 + w_y^2}$.

**Return type**
float

**write_mode**(*prefix=''*, *n_points=501*, *field_type=FieldType.EM_E*)

**write_mode_1D**(*s_cut*, *val1*, *val2=None*, *n_points=501*, *prefix=''*, *field_type=FieldType.EM_E*)

**wx**()

Returns the $x$ component moment of the second moment width.

**Return type**
float

**wy**()

Returns the $y$ component moment of the second moment width.

**Return type**
float

**class** modes.**ModeAC**(*sim*, *m*)

Bases: *Mode*

Class representing a single acoustic (AC) mode.

**class** modes.**ModeEM**(*sim*, *m*)

Bases: *Mode*

Class representing a single electromagnetic (EM) mode.

**class** modes.**ModeInterpolator**(*simresult*)

Bases: object

Helper class for plotting modes. Factors common info from Simulation that each mode can draw on, but we only need to do once for each Sim.

**cleanup**()

**define_plot_grid_1D**(*s_cut*, *val1*, *val2*, *n_pts*)

**define_plot_grid_2D**(*n_pts=501*)

Define interpolation plotting grids for a nominal n_pts**2 points distributed evenly amongst x and y.

**interpolate_mode_i**(*md*, *field_type*, *dims=2*)

**zero_arrays**()

## 13.5 structure module

The `structure` module provides functions for defining and constructing waveguides. The diagrams in Chapter 2 can be used to identify which parameters (`slab_a_x, slab_c_y, material_d` etc) correspond to each region.

**class** structure.**Structure**(**largs*, ***kwargs*)

Bases: `object`

Represents the geometry and material properties (elastic and optical) of a waveguide structure.

**Parameters**

- **domain_x** (*float*) – The horizontal period of the unit cell in nanometers.
- **domain_y** (*float*) – The vertical period of the unit cell in nanometers. If None, domain_y = domain_x.
- **inc_shape** (*str*) –

  **Shape of inclusions that have template mesh, currently:**
  `circular` `rectangular` `slot` `rib` `slot_coated` `rib_coated` `rib_double_coated` `pedestal` `onion` `onion2` `onion3`. rectangular is default.
- **symmetry_flag** (*bool*) – True if materials all have sufficient symmetry that their tensors contain only 3 unique values. If False must specify full [3,3,3,3] tensors.
- **material_bkg** (`Material`) – The outer background material.
- **material_a** (`Material`) – The primary inclusion material.
- **material_b-r** (`Material`) – Materials of additional layers.
- **loss** (*bool*) – If False, Im(n) = 0, if True n as in `Material` instance.
- **lc_bkg** (*float*) – Length constant of meshing of background medium (smaller = finer mesh)
- **lc_refine_1** (*float*) – factor by which lc_bkg will be reduced on inclusion surfaces; lc_surface = lc_bkg / lc_refine_1. Larger lc_refine_1 = finer mesh.
- **lc_refine_2-6'** (*float*) – factor by which lc_bkg will be reduced on chosen surfaces; lc_surface = lc_bkg / lc_refine_2. see relevant .geo files.

**calc_AC_modes**(*num_modes*, *q_AC*, *shift_Hz=None*, *EM_sim=None*, *bcs=None*, *debug=False*, ***args*)

Run a simulation to find the Structure's acoustic modes.

**Parameters**

**num_modes** (*int*) – Number of AC modes to solve for.

**Keyword Arguments**

- **q_AC** (*float*) – Wavevector of AC modes.
- **shift_Hz** (*float*) – Guesstimated frequency of modes, will be origin of FEM search. NumBAT will make an educated guess if shift_Hz=None. (Technically the shift and invert parameter).
- **EM_sim** (`EMSimResult` object) – Typically an acoustic simulation follows on from an optical one. Supply the `EMSimResult` object so the AC FEM mesh can be constructed from this. This is done by removing vacuum regions.

**Returns**
ACSimResult object

**calc_EM_modes**(*num_modes*, *wl_nm*, *n_eff*, *Stokes=False*, *debug=False*, ***args*)

Run a simulation to find the Structure's EM modes.

**Parameters**

- **num_modes** (*int*) – Number of EM modes to solve for.
- **wl_nm** (*float*) – Wavelength of EM wave in vacuum in nanometres.
- **n_eff** (*float*) – Guesstimated effective index of fundamental mode, will be origin of FEM search.

**Returns**
EMSimResult object

**check_mesh**()

Visualise geometry and mesh with gmsh.

**get_material**(*k*)

**get_optical_materials**()

**get_structure_plotter_acoustic_velocity**(*vel_index=0*, *n_points=500*)

Make plotter for arbitrary 1D and 2D slices of the elastic acoustic phase speed.

**Parameters**
**vel_index** (*0,1,2*) – Index of the elastic mode phase speed to plot.

Currently only works for isotropic materials.

**get_structure_plotter_epsilon**(*n_points=500*)

Make plotter for arbitrary 1D and 2D slices of the dielectric constant profile.

**get_structure_plotter_refractive_index**(*n_points=500*)

Make plotter for arbitrary 1D and 2D slices of the refractive index profile.

**get_structure_plotter_stiffness**(*c_I*, *c_J*, *n_points=500*)

Make plotter for arbitrary 1D and 2D slices of the elastic stiffness.

**plot_mail_mesh**(*outpref*)

Visualise the mesh in .mail format.

**plot_mesh**(*outpref*, *combo_plot=True*)

Visualise mesh with gmsh and save to a file.

If combo_plot==False, wire frame and mesh node plots are stored in separate files.

**plot_refractive_index_profile**(*pref*)

Draw 2D plot of refractive index profile.

**plot_refractive_index_profile_rough**(*prefix*, *n_points=200*, *as_epsilon=False*)

Draws refractive index profile by primitive sampling, not proper triangular mesh sampling

**set_xyshift_ac**(*x*, *y*)

**set_xyshift_em**(*x*, *y*)

**using_curvilinear_elements**()

**using_linear_elements**()

## 13.6 modecalcs module

The `modecalcs` module is responsible for the core engine to construct and solve the optical and elastic finite-element problems.

**class** modecalcs.**ACSimulation**(*structure*, *num_modes=20*, *shift_Hz=None*, *q_AC=0*, *simres_EM=None*, *calc_AC_mode_power=False*, *debug=False*)

Bases: *Simulation*

**calc_acoustic_losses**(*fixed_Q=None*)

**calc_field_energies**()

**calc_field_powers**()

**calc_modes**(*bcs=None*)

Run a Fortran FEM calculation to find the acoustic modes.

Returns a `Simulation` object that has these key values:

eigs_nu: a 1D array of Eigenvalues (frequencies) in [1/s]

**fem_evecs: the associated Eigenvectors, ie. the fields, stored as** [field comp, node num on element, Eig value, el num]

AC_mode_energy: the elastic power in the acoutic modes.

**choose_eigensolver_frequency_shift**()

**do_main_eigensolve**(*bcs*)

**is_AC**()

Returns true if the solver is setup for an acoustic problem.

**make_result**()

**class** modecalcs.**EMSimulation**(*structure*, *num_modes=20*, *wl_nm=1550*, *n_eff_target=None*, *Stokes=False*, *calc_EM_mode_energy=True*, *debug=False*, ***args*)

Bases: *Simulation*

**calc_field_energies**()

**calc_field_powers**()

**calc_modes**(*shortrun*)

Run a Fortran FEM calculation to find the optical modes.

Returns a `Simulation` object that has these key values:

eigs_kz: a 1D array of Eigenvalues (propagation constants) in [1/m]

fem_evecs: the associated Eigenvectors, ie. the fields, stored as [field comp, node nu on element, Eig value, el nu]

**EM_mode_power: the power in the optical modes. Note this power is negative for modes travelling in the negative** z-direction, eg the Stokes wave in backward SBS.

**convert_to_Stokes**()

**do_main_eigensolve**(*shortrun*)

**is_EM**()

Returns true if the solver is setup for an electromagnetic problem.

**make_result**()

class modecalcs.**Simulation**(*structure*, *num_modes*, *debug*)

Bases: object

Class for calculating the electromagnetic and/or acoustic modes of a Struct object.

**clean_for_pickle**()

**get_sim_result**()

**is_AC**()

Returns true if the solver is setup for an acoustic problem.

**is_EM**()

Returns true if the solver is setup for an electromagnetic problem.

**static load_simulation**(*prefix*)

**save_simulation**(*prefix*)

modecalcs.**ac_mode_calculation**(*wg*, *num_modes*, *q_AC*, *shift_Hz*, *EM_sim*, *bcs*, *debug*, ***args*)

modecalcs.**bkwd_Stokes_modes**(*sim*)

modecalcs.**em_mode_calculation**(*wg*, *num_modes*, *wl_nm*, *n_eff*, *Stokes*, *debug*, ***args*)

## 13.7 integration module

The integration module is responsible for calculating gain and loss information from existing mode data. Integration and gain calculation routines for NumBAT.

This module provides classes and functions for calculating SBS gain, Q-factors, and related quantities from electromagnetic and acoustic simulation results. It includes the GainProps class for storing and accessing gain and loss data, as well as functions for performing the necessary integrals and interpolations.

class integration.**GainProps**

Bases: object

Stores and manages SBS gain, Q-factor, and related properties for a set of EM and acoustic modes.

**_max_pumps_m**

Maximum number of pump EM modes.

**Type**
int

**_max_Stokes_m**

Maximum number of Stokes EM modes.

**Type**
int

**_max_ac_m**

Maximum number of acoustic modes.

**Type**
int

**_allowed_pumps_m**

Indices of allowed pump EM modes.

**Type**
list

**_allowed_Stokes_m**

Indices of allowed Stokes EM modes.

**Type**
list

**_allowed_ac_m**

Indices of allowed acoustic modes.

**Type**
list

**_gain_tot**

Total SBS gain array.

**Type**
np.ndarray

**_gain_PE**

Photoelastic SBS gain array.

**Type**
np.ndarray

**_gain_MB**

Moving boundary SBS gain array.

**Type**
np.ndarray

**def_m_pump**

Default pump mode index.

**Type**
int

**def_m_Stokes**

Default Stokes mode index.

**Type**
int

**linewidth_Hz**

Acoustic linewidth(s) in Hz.

**Type**
float or np.ndarray

**alpha**

Acoustic loss(es).

**Type**
float or np.ndarray

**Q_factor**

Acoustic Q-factor(s).

**Type**
float or np.ndarray

**sim_AC**

Acoustic simulation object.

**Q_factor_all()**

Return the Q-factor for all modes.

**alpha_all**()

Return the acoustic loss (alpha) for all modes.

**check_acoustic_expansion_size**()

Check if the maximum gain occurs near the upper end of the acoustic mode range. Warn if so, as this may indicate the need for more acoustic modes.

**gain_MB**(*m_AC*)

Return the moving boundary SBS gain for the current pump, Stokes, and given acoustic mode index.

**gain_MB_all**()

Return the moving boundary SBS gain for the current pump and Stokes over all acoustic modes.

**gain_MB_all_by_em_modes**(*m_pump*, *m_Stokes*)

Return the moving boundary SBS gain for given pump and Stokes indices over all acoustic modes.

**gain_MB_raw**()

Return the raw moving boundary SBS gain array.

**gain_PE**(*m_AC*)

Return the photoelastic SBS gain for the current pump, Stokes, and given acoustic mode index.

**gain_PE_all**()

Return the photoelastic SBS gain for the current pump and Stokes over all acoustic modes.

**gain_PE_all_by_em_modes**(*m_pump*, *m_Stokes*)

Return the photoelastic SBS gain for given pump and Stokes indices over all acoustic modes.

**gain_PE_raw**()

Return the raw photoelastic SBS gain array.

**gain_total**(*m_AC*)

Return the total SBS gain for the current pump, Stokes, and given acoustic mode index.

**gain_total_all**()

Return the total SBS gain for the current pump and Stokes over all acoustic modes.

**gain_total_all_by_em_modes**(*m_pump*, *m_Stokes*)

Return the total SBS gain for given pump and Stokes indices over all acoustic modes.

**gain_total_raw**()

Return the raw total SBS gain array.

**linewidth_Hz_all**()

Return the linewidth (Hz) for all modes.

**plot_spectra**(*freq_min=0.0*, *freq_max=50000000000.0*, *num_interp_pts=0*, *dB=False*, *dB_peak_amp=10*, *mode_comps=False*, *logy=False*, *save_txt=False*, *prefix=''*, *suffix=''*, *decorator=None*, *show_gains='All'*, *mark_modes_thresh=0.02*)

Plot SBS gain spectra using the stored gain arrays and simulation data.

**Parameters**

- **freq_min** (*float*) – Minimum frequency for plot (Hz).
- **freq_max** (*float*) – Maximum frequency for plot (Hz).
- **num_interp_pts** (*int*) – Number of interpolation points.
- **dB** (*bool*) – Plot in dB scale if True.
- **dB_peak_amp** (*float*) – Peak amplitude for dB plot.
- **mode_comps** (*bool*) – Plot mode components if True.
- **logy** (*bool*) – Logarithmic y-axis if True.

- **save_txt** (*bool*) – Save data to text file if True.
- **prefix** (*str*) – Filename prefix for output.
- **suffix** (*str*) – Filename suffix for output.
- **decorator** (*callable*) – Function to decorate the plot.
- **show_gains** (*str*) – Which gains to show ('All' or specific).
- **mark_modes_thresh** (*float*) – Threshold for marking modes.

**Returns**
Output of plotgain.plot_gain_spectra().

**set_EM_modes**(*mP*, *mS*)

Set the default pump and Stokes EM mode indices.

**set_allowed_AC**(*m_allow*)

Set allowed acoustic modes.

**set_allowed_EM_Stokes**(*m_allow*)

Set allowed Stokes EM modes.

**set_allowed_EM_pumps**(*m_allow*)

Set allowed pump EM modes.

integration.**gain_and_qs**(*simres_EM_pump*, *simres_EM_Stokes*, *simres_AC*, *q_AC*, *EM_mode_index_pump=0*, *EM_mode_index_Stokes=0*, *AC_mode_index=0*, *fixed_Q=None*, *typ_select_out=None*, *new_call_format=False*)

Calculate interaction integrals and SBS gain.

Implements Eqs. 33, 41, 45, 91 of Wolff et al. PRA 92, 013836 (2015) doi/10.1103/PhysRevA.92.013836 These are for Q_photoelastic, Q_moving_boundary, the Acoustic loss "alpha", and the SBS gain respectively.

Note there is a sign error in published Eq. 41. Also, in implementing Eq. 45 we use integration by parts, with a boundary integral term set to zero on physical grounds, and filled in some missing subscripts. We prefer to express Eq. 91 with the Lorentzian explicitly visible, which makes it clear how to transform to frequency space. The final integrals are

$$Q^{\mathrm{PE}} = -\varepsilon_0 \int_A \mathrm{d}^2 r \sum_{ijkl} \varepsilon_r^2 e_i^{(s)\star} e_j^{(p)} p_{ijkl} \partial_k u_l^*,$$

$$Q^{\mathrm{MB}} = \int_C \mathrm{d}\mathbf{r} (\mathbf{u}^* \cdot \hat{\mathrm{n}}) \big[ (\varepsilon_a - \varepsilon_b)\varepsilon_0 (\hat{\mathrm{n}} \times \mathbf{e}) \cdot (\hat{\mathrm{n}} \times \mathbf{e}) - (\varepsilon_a^{-1} - \varepsilon_b^{-1})\varepsilon_0^{-1} (\hat{\mathrm{n}} \cdot \mathbf{d}) \cdot (\hat{\mathrm{n}} \cdot \mathbf{d}) \big],$$

$$\alpha = \frac{\Omega^2}{\mathcal{E}_{ac}} \int \mathrm{d}^2 r \sum_{ijkl} \partial_i u_j^* \eta_{ijkl} \partial_k u_l,$$

$$\Gamma = \frac{2\omega\Omega \mathrm{Re}(Q_1 Q_1^*)}{P_p P_s \mathcal{E}_{ac}} \frac{1}{\alpha} \frac{\alpha^2}{\alpha^2 + \kappa^2}.$$

**Parameters**

- **sim_EM_pump** (Simulation object) – Contains all info on pump EM modes
- **sim_EM_Stokes** (Simulation object) – Contains all info on Stokes EM modes
- **sim_AC** (Simulation object) – Contains all info on AC modes
- **q_AC** (*float*) – Propagation constant of acoustic modes.

**Keyword Arguments**

- **EM_mode_index_pump** (*int/string*) – Specify mode number of EM mode 1 (pump mode) to calculate interactions for. Numbering is python index so runs from 0 to num_EM_modes-1, with 0 being fundamental mode (largest prop constant). Can also set to 'All' to include all modes.
- **EM_mode_index_Stokes** (*int/string*) – Specify mode number of EM mode 2 (stokes mode) to calculate interactions for. Numbering is python index so runs from 0 to num_EM_modes-1, with 0 being fundamental mode (largest prop constant). Can also set to 'All' to include all modes.
- **AC_mode_index** (*int/string*) – Specify mode number of AC mode to calculate interactions for. Numbering is python index so runs from 0 to num_AC_modes-1, with 0 being fundamental mode (largest prop constant). Can also set to 'All' to include all modes.
- **fixed_Q** (*int*) – Specify a fixed Q-factor for the AC modes, rather than calculating the acoustic loss (alpha).

**Returns**

**The SBS gain including both photoelastic and moving boundary contributions.**
Note this will be negative for backwards SBS because gain is expressed as gain in power as move along z-axis in positive direction, but the Stokes waves experience gain as they propagate in the negative z-direction. Dimensions = [n_modes_EM_Stokes,n_modes_EM_pump,n_modes_AC].

**SBS_gain_PE**
[The SBS gain for only the photoelastic effect.] The comment about negative gain (see SBS_gain above) holds here also. Dimensions = [n_modes_EM_Stokes,n_modes_EM_pump,n_modes_AC].

**SBS_gain_MB**
[The SBS gain for only the moving boundary effect.] The comment about negative gain (see SBS_gain above) holds here also. Dimensions = [n_modes_EM_Stokes,n_modes_EM_pump,n_modes_AC].

alpha : The acoustic power loss for each mode in [1/s]. Dimensions = [n_modes_AC].

**Return type**
SBS_gain

integration.**get_gains_and_qs**(*sim_EM_pump*, *sim_EM_Stokes*, *sim_AC*, *q_AC*, *EM_mode_index_pump=0*, *EM_mode_index_Stokes=0*, *AC_mode_index=0*, *fixed_Q=None*, *typ_select_out=None*, ***kwargs*)

integration.**symmetries**(*simres*, *n_points=10*, *negligible_threshold=1e-05*)

Plot EM mode fields.

**Parameters**
**simres** – A Struct instance that has had calc_modes calculated

**Keyword Arguments**
**n_points** (*int*) – The number of points across unitcell to interpolate the field onto.

## 13.8 plotting module

The plotting module is responsible for generating all standard graphs.

CHAPTER

# FOURTEEN

# REFERENCES


[1] Björn C. P. Sturmberg, Kokou Bertin Dossou, Michael J. A. Smith, Blair Morrison, Christopher G. Poulton, and Michael J. Steel. Finite element analysis of stimulated brillouin scattering in integrated photonic waveguides. *J. Lightwave Technol.*, 37(15):3791–3804, Aug 2019. URL: https://opg.optica.org/jlt/abstract.cfm?URI=jlt-37-15-3791.

[2] C Wolff, B Stiller, B J Eggleton, M J Steel, and C G Poulton. Cascaded forward brillouin scattering to all stokes orders. *New Journal of Physics*, 19(2):023021, feb 2017. URL: https://dx.doi.org/10.1088/1367-2630/aa599e, doi:10.1088/1367-2630/aa599e.

[3] Christian Wolff, Christopher G. Poulton, Michael J. Steel, and Gustavo Wiederhecker. Chapter two - theoretical formalisms for stimulated brillouin scattering. In Benjamin J. Eggleton, Michael J. Steel, and Christopher G. Poulton, editors, *Brillouin Scattering Part 1*, volume 109 of Semiconductors and Semimetals, pages 27–91. Elsevier, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0080878422000023, doi:https://doi.org/10.1016/bs.semsem.2022.04.002.

[4] Kokou Dossou and Marie Fontaine. A high order isoparametric finite element method for the computation of waveguide modes. *Computer Methods in Applied Mechanics and Engineering*, 194(6):837–858, 2005. URL: https://www.sciencedirect.com/science/article/pii/S0045782504003032, doi:https://doi.org/10.1016/j.cma.2004.06.011.

[5] A.-C. Hladky-Hennion, P. Langlet, and M. de Billy. Finite element analysis of the propagation of acoustic waves along waveguides immersed in water. *Journal of Sound and Vibration*, 200(4):519–530, 1997. URL: https://www.sciencedirect.com/science/article/pii/S0022460X9690749X, doi:https://doi.org/10.1006/jsvi.1996.0749.


indices:indices-and-tables

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX

## Symbols



## A



## B



## C



## D

## E



## F



## G



## H



## I



## K



## L

## M



## N



## O



## P



## Q



## R