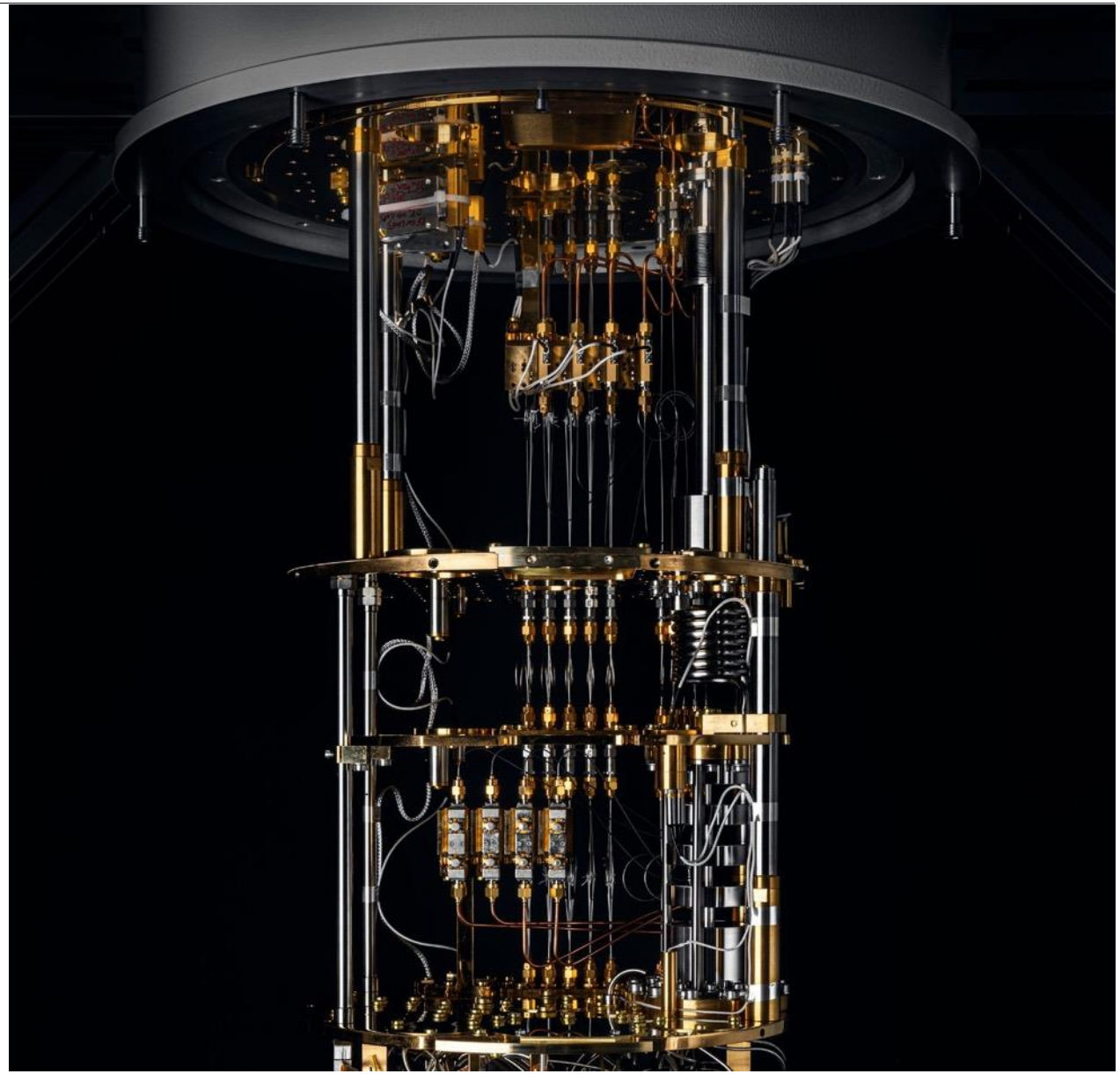
KQCircuits: Simulations

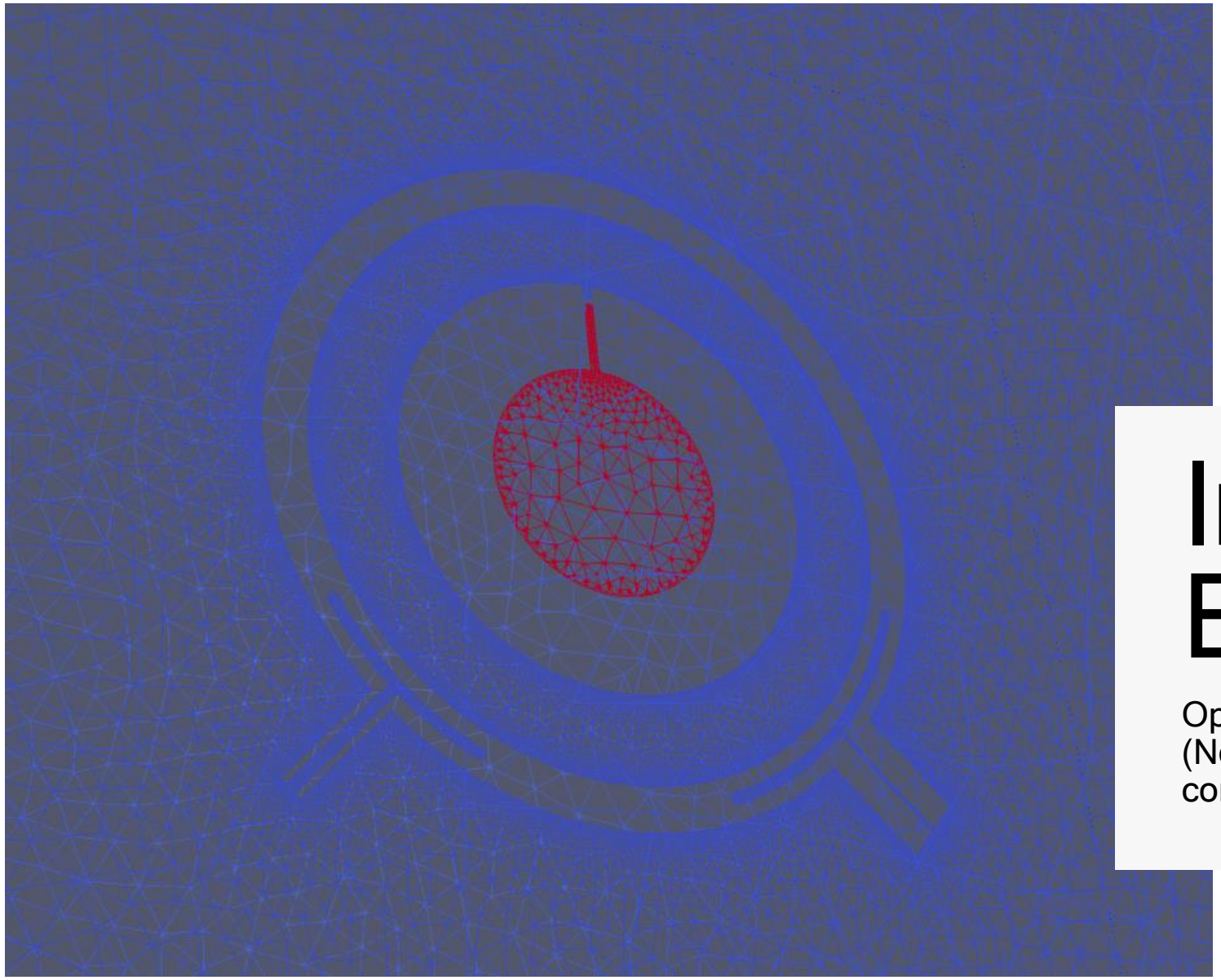


Niko Savola, Quantum Engineer Intern

WE BUILD QUANTUM COMPUTERS

Overview

- Open-source simulations with Elmer
- Demo
  - Ansys support
  - Cross-section simulations
  - Loss-function optimiser
- Simulation classes in KQC
  - Parameter sweeps
- Qubit- resonator properties with scQubits



Elmer FEM

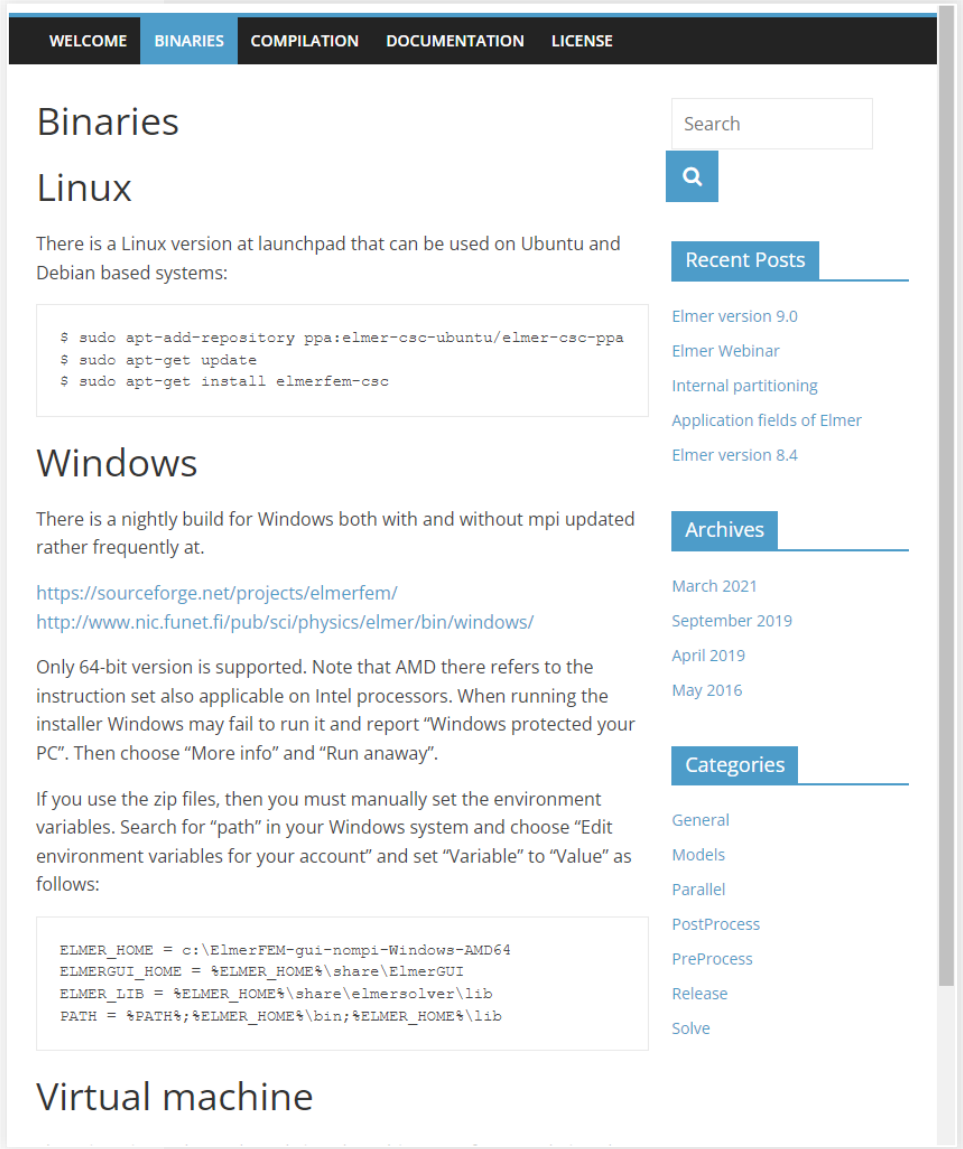
open source multiphysical simulation software

Installing Elmer

Open-source FEM solver developed at CSC (Non-profit state enterprise for scientific computing in Finland)

Preparing for simulations

- Think where the simulations will be run
  - Remote Linux server, locally on Windows?
- Required open-source software
  - gmsh, for meshing
    - Should work by pip install gmsh
  - Elmer, for finite-element
    - Download and install binaries at elmerfem.org/blog/binaries/
  - ParaView, for viewing results
- gmsh and ParaView support Linux, macOS and Windows. Elmer provides binaries only for Debian Linux and Windows (but other OSs may be compiled)



IQM

Installing Elmer

This likely is the most unclear, so we'll walk through a typical Windows installation.

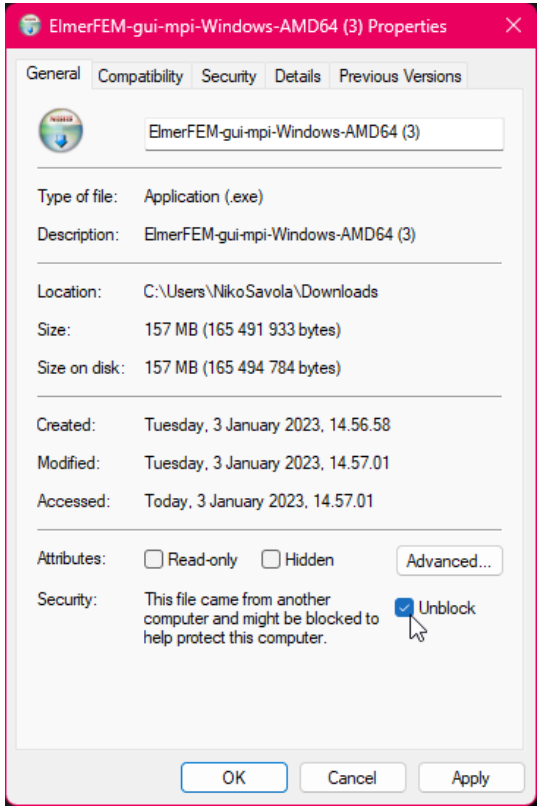
1. Download the gui-mpi version from nic.funet.fi/pub/sci/physics/elmer/bin/windows

Index of /pub/sci/physics/elmer/bin/windows

Name	Last modified	Size	Description
Parent Directory		-	
old-gcc9/	2021-04-06 15:07	-	
rel9.0/	2022-08-02 11:45	-	
scripts/	2021-03-11 15:44	-	
ElmerFEM-gui-mpi-Windows-AMD64.exe	2023-01-03 03:19	158M	
ElmerFEM-gui-mpi-Windows-AMD64.zip	2023-01-03 03:19	218M	
ElmerFEM-gui-nompi-Windows-AMD64.exe	2023-01-03 03:16	151M	
ElmerFEM-gui-nompi-Windows-AMD64.zip	2023-01-03 03:16	209M	
ElmerFEM-nogui-mpi-Windows-AMD64.exe	2023-01-03 03:08	93M	
ElmerFEM-nogui-mpi-Windows-AMD64.zip	2023-01-03 03:08	129M	
ElmerFEM-nogui-nompi-Windows-AMD64.exe	2023-01-03 03:02	88M	
ElmerFEM-nogui-nompi-Windows-AMD64.zip	2023-01-03 03:02	123M	
ElmerFEM-tests.tar.gz	2021-04-06 14:46	25M	
Readme1st.txt	2021-03-08 21:26	2.6K	
RunElmer.bat	2021-12-01 21:32	164	

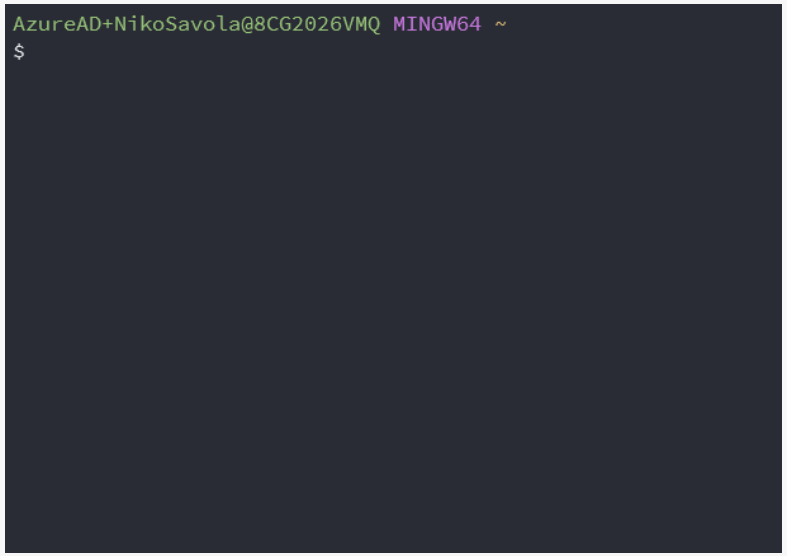
2. You likely need to *Unblock* the installer as it is not certified

3. Run the installer



Installing Elmer

Test that Elmer works by running ElmerSolver in terminal



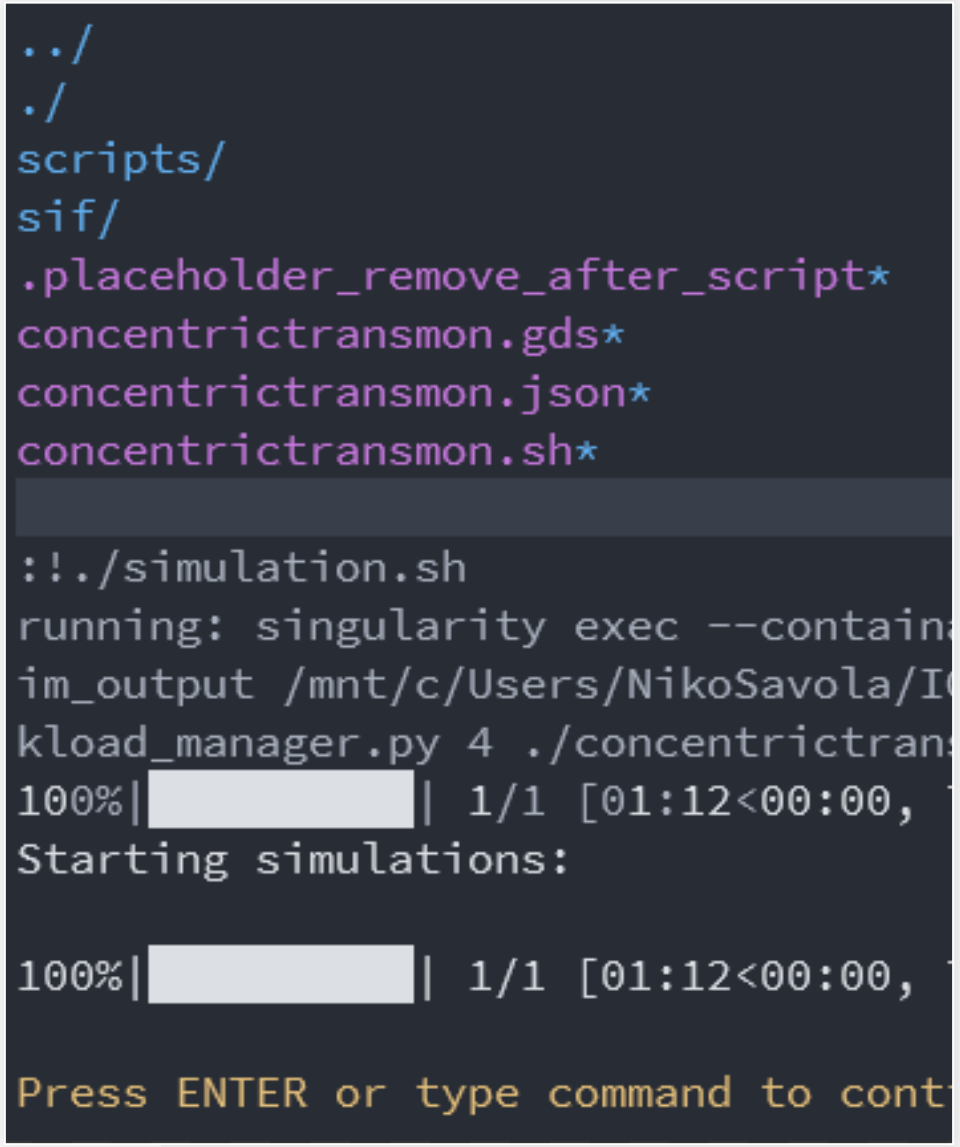
The simulations currently require running .sh files. To this end, we recommend using Git Bash.

Windows Subsystem for Linux (WSL) may be used as well. In this case, use Ubuntu WSL and follow the Debian instructions on the Elmer website:

```
sudo apt-add-repository \
 ppa:elmer-csc-ubuntu/elmer-csc-ppa
sudo apt-get update
sudo apt-get install elmerfem-csc
```

Alternative: Singularity container

- Similar to Docker images but work better in HPC environments
- Pre-compiled container including KQCircuits, gmsh and Elmer provided at oras://ghcr.io/iqm-finland/kqcircuits:main-singularity
- More info at [KQCircuits docs → Developer Guide → Containers → Singularity usage]



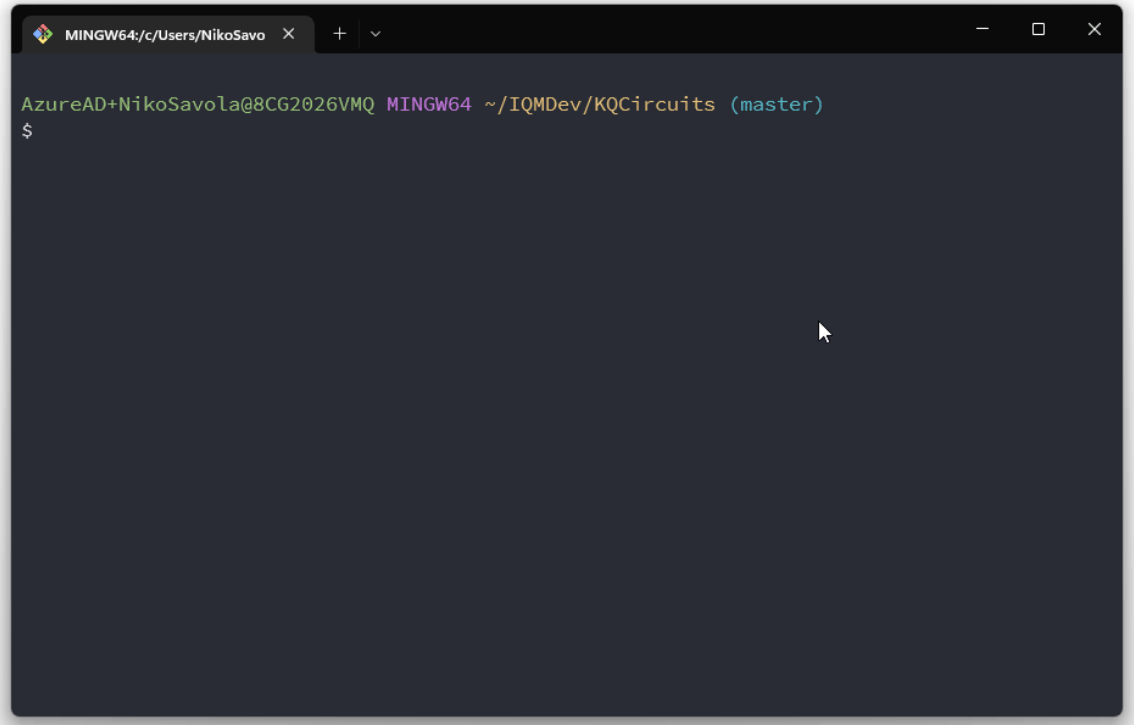
Running Simulations

As an example, let's run a **simulation script** generating 4 simulations and run them in parallel.

From the root folder for KQC, run (or execute the file in your IDE)

python klayout_package/python/scripts/simulations/simple_workload_manager_example.py

This should open KLayout with the exported geometries, you may **close this**.



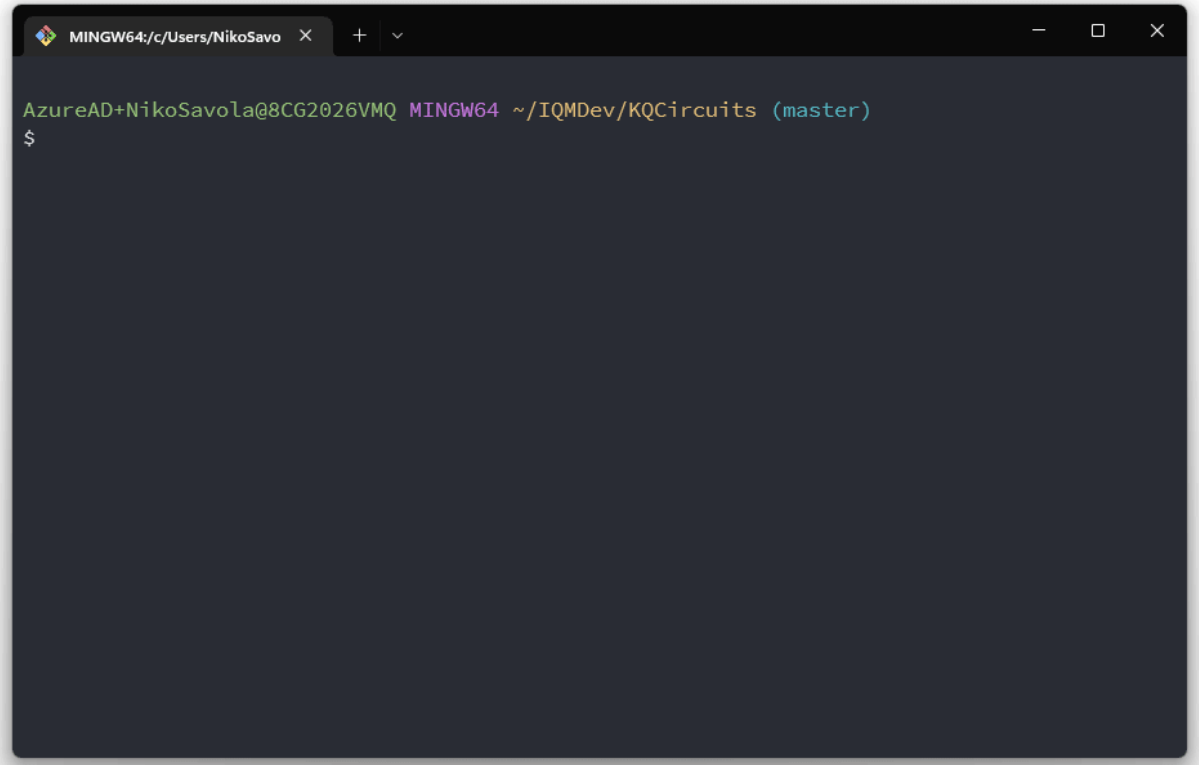
Running Simulations

Then go to your **KQC tmp** folder (KQCircuits/tmp by default). This may be overwritten with the KQC_TMP_PATH env.

Locate the **script output folder** simple_workload_manager_example_output and enter it.

All of the exported simulations are **run** (in parallel) **by executing** ./simulation.sh

By default, the mesh is shown in gmsh before simulating. These windows need to be closed to continue.

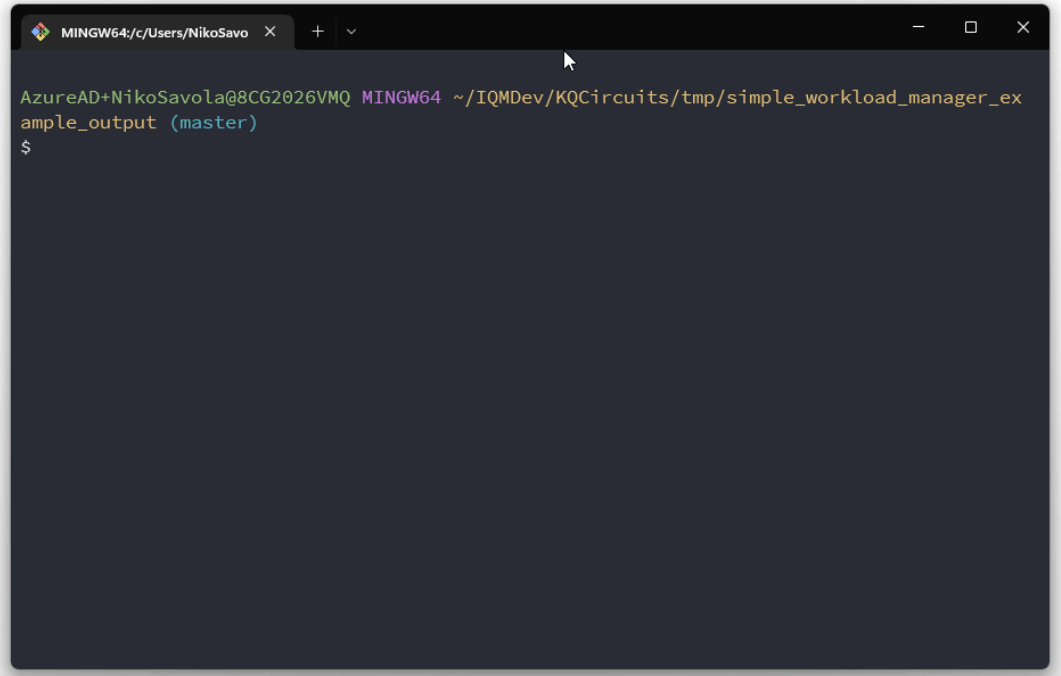


Viewing the results

We just did a capacitance simulation of three qubits with readout- and drivelines each. This provides a 9x9 capacitance matrix for examining coupling.

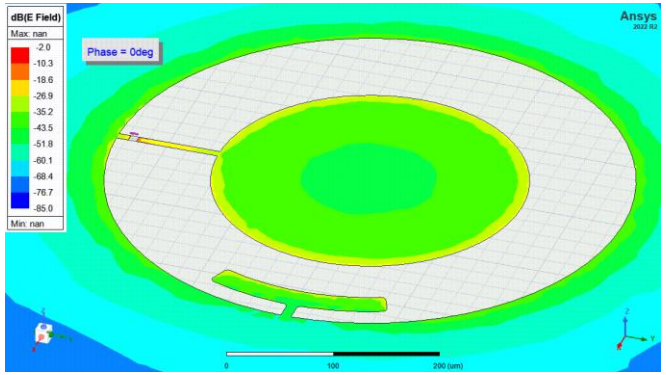
The matrix is stored in JSON form in corresponding _..._project_results.json files for each geometry variation.

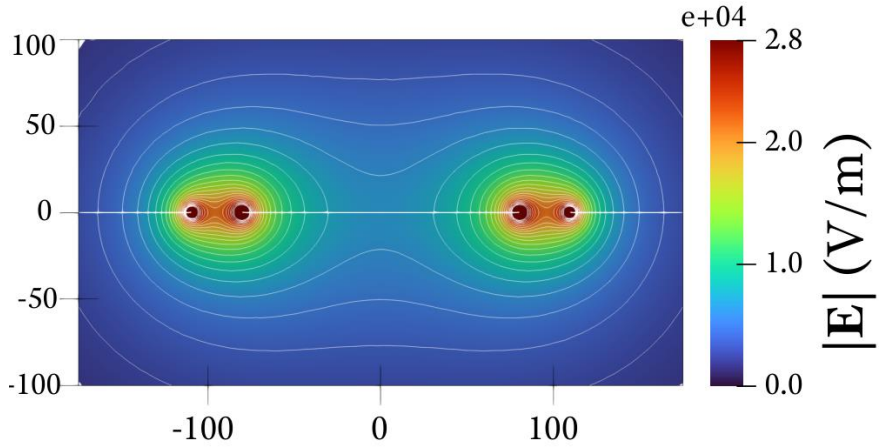
ParaView may be used to see the field solutions. Although here it isn't very interesting w/o enabling saving the electric field in Elmer. The field solution is stored in a .vtu file inside the individual simulation folders. Use the terminal with paraview or File → Open inside ParaView.



Other out-of-the-box simulations

Elmer FEM
open source multiphysical simulation software





- Ansys Q3D with ports converted to sources
- Ansys HFSS with ports
- Ansys HFSS Eigenmode with ideal inductors modelling junctions/SQUIDs
  - pyEPR support
- 2D cross-sections in Elmer
  - Kinetic inductance for some penetration depth λ
  - Energy participation ratio (EPR) [1]
- Sonnet

[1] Z. K. Minev et al., 'Energy-participation quantization of Josephson circuits', *npj Quantum Inf*, vol. 7, no. 1, Art. no. 1, Aug. 2021

Running Simulations in Ansys

Essentially, the user interface for running simulation batches is the same as Elmer:

1. Export simulation output folder with a Python script, but use the `export_ansys` function instead `of export_elmer`
2. Run the simulation.bat (Windows only due to being the main Ansys platform but easy to edit for Linux). This will import the geometry, setup the simulation and run the variations serially.
3. For Q3D, `..._project_results.json` files in the same format are generated

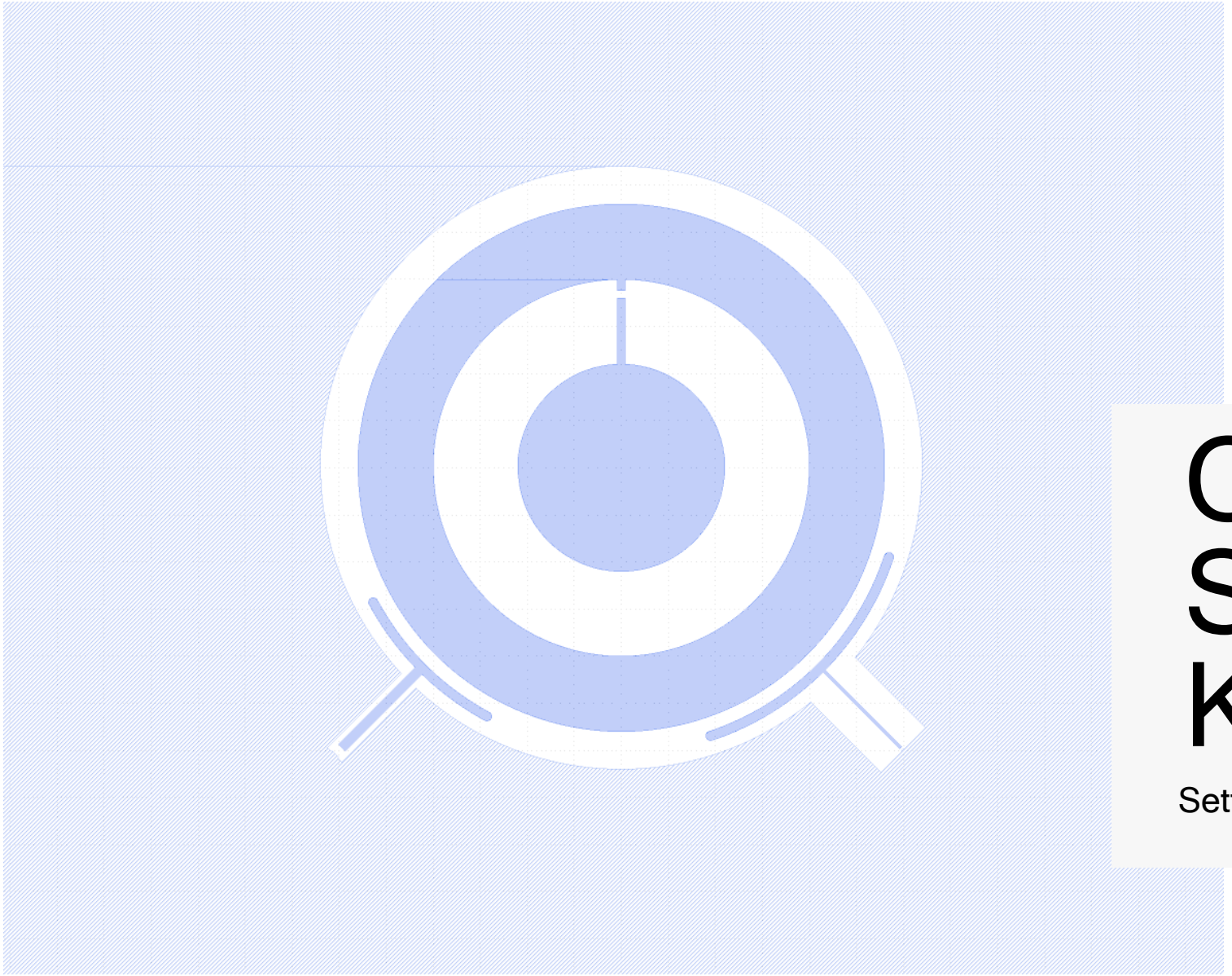
Note:

The API for the export functions differ a bit, as Ansys uses a different logic for *adaptive meshing* and convergence criteria by default.

```
export_parameters_ansys = {
    'ansys_tool': 'q3d',
    'path': dir_path,
    'exit_after_run': True,
    'percent_error': 0.3,
    'maximum_passes': 18,
    'minimum_passes': 2,
    'minimum_converged_passes': 2,
}
```

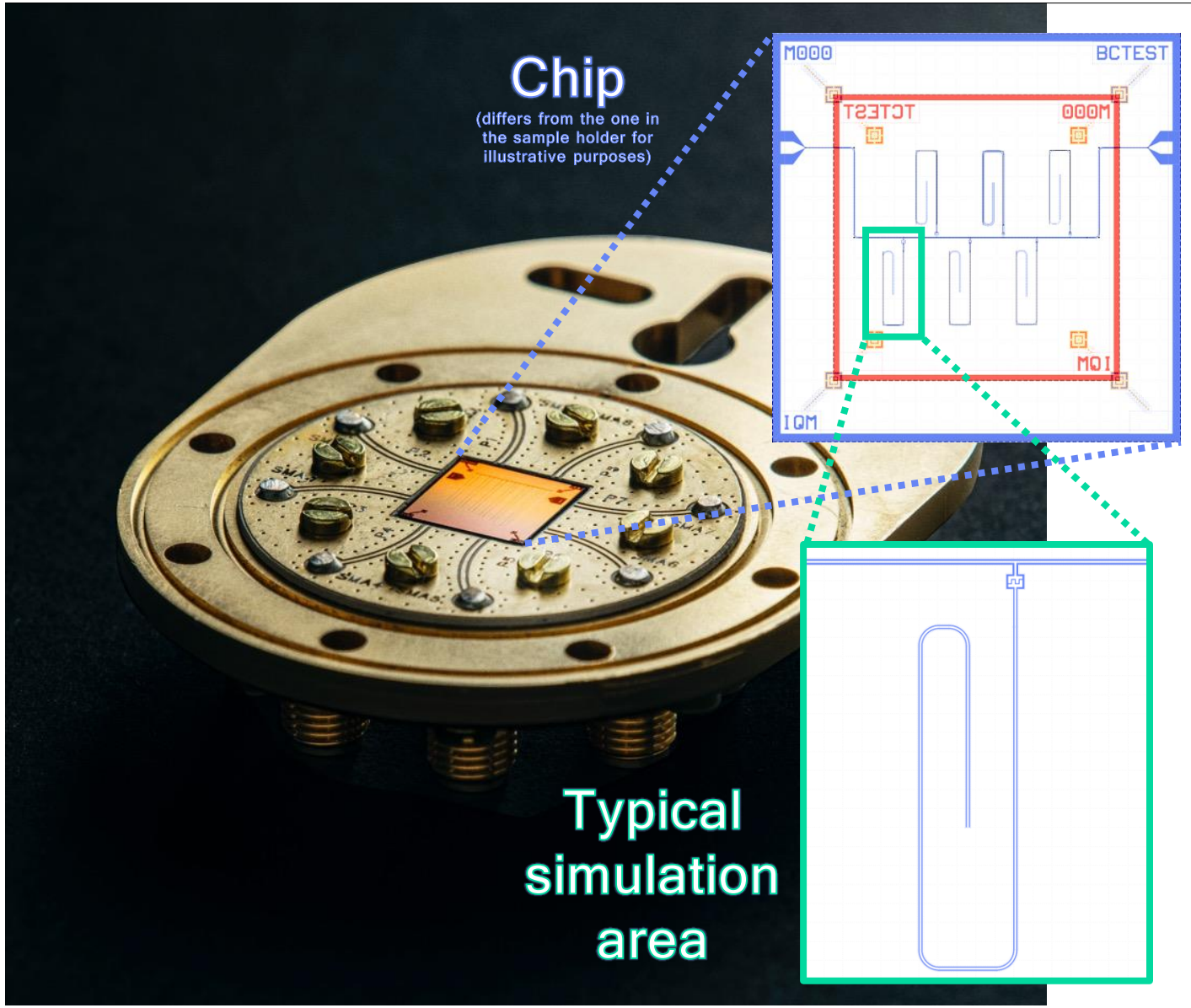
```
export_parameters_elmer = {
    'tool': 'capacitance',
    'workflow': {
        'python_executable': 'python',
        'n_workers': 4,
        'elmer_n_processes': 4,
        'gmsh_n_threads': 4,
    },
    'mesh_size': {
        'global_max': 50.,
        'gap&signal': [2., 4.],
        'gap&ground': [2., 4.],
        'port': [1., 4.],
    }
}
```

These differences may be seen in, for example, [double_pads_sim.py]



Creating Simulations in KQCircuits

Setting up simulations



Creating Simulations

Let's create our own simulations now. Designing the geometry is most of the work. Subsequently, we need to

1. Create a `Simulation class` from the geometry
   - Geometry should be as limited as possible for computational feasibility
   - Add ports where applicable
2. Create separate simulation script
3. Set the simulation settings
4. Export!

```python
from kqcircuits.qubits.concentric_transmon import ConcentricTransmon
from kqcircuits.simulations.simulation import Simulation
from kqcircuits.pya_resolver import pya
from kqcircuits.util.parameters import Param, pdt, add_parameters_from
from kqcircuits.simulations.port import InternalPort


@add_parameters_from(ConcentricTransmon, "*", junction_type="Sim", fluxline_type="none")
class ConcentricTransmonCouplingsSim(Simulation):
    qubit_faces = Param(pdt.TypeList, "List of faces", ["1t1", "2b1"])

    def build(self):
        # Translation between elements faces and port faces
        port_face = self.face_ids.index(self.qubit_faces[0])

        # Insert the qubit
        qubit_cell = self.add_element(ConcentricTransmon, **{**self.get_parameters(),
'face_ids': self.qubit_faces})
        qubit_trans = pya.DTrans(0, False, (self.box.left + self.box.right) / 2,
(self.box.bottom + self.box.top) / 2)
        qubit_inst, refp = self.insert_cell(qubit_cell, qubit_trans, rec_levels=None)

        # Add ports at the two islands
        if self.junction_type == 'Sim':
            self.ports.append(
                InternalPort(1, refp["port_squid_a"], face=port_face))
            self.ports.append(
                InternalPort(2, refp["port_squid_b"], face=port_face))

        # Add ports at the couplers
        for i in range(len(self.couplers_angle)):
            self.produce_waveguide_to_port(refp[f"port_coupler_{i+1}"],
refp[f"port_coupler_{i+1}_corner"], i+3,
                                           'bottom', a=self.couplers_a[i],
b=self.couplers_b[i], face=port_face)
```

Create a Simulation class

Geometry can be inserted, created, and combined as normal with the KLayout API.

Now we additionally add *internal* or *edge* ports with

```
self.ports.append(InternalPort(i,
        refpoint, face))
```

or for connections to edges of simulation area

```
self.produce_waveguide_to_port(
        location, direction, edge[str])
```

Parametrization is copied from the qubit with `@add_parameters_from`

```python
import logging
import sys
from pathlib import Path

from kqcircuits.simulations.concentrictransmon_couplings_sim import
ConcentricTransmonCouplingsSim
from kqcircuits.pya_resolver import pya
from kqcircuits.simulations.export.elmer.elmer_export import export_elmer
from kqcircuits.simulations.export.simulation_export import export_simulation_oas,
sweep_simulation, \
    cross_sweep_simulation
from kqcircuits.util.export_helper import create_or_empty_tmp_directory,
get_active_or_new_layout, \
    open_with_klayout_or_default_application

# Prepare output directory
dir_path = create_or_empty_tmp_directory(Path(__file__).stem + "_output")

# Simulation parameters
sim_class = ConcentricTransmonCouplingsSim
sim_parameters = {
    # Arguments for the base Simulation class
    'name': 'concentrictransmon',
    'box': pya.DBox(pya.DPoint(0, 0), pya.DPoint(2000, 2000)),  # total area for simulation
    'use_ports': True,
    'use_internal_ports': True,  # wave ports are actually internal (lumped) ports instead of
at the edge
    'waveguide_length': 100,  # wave port length before terminating with InternalPort in this
case

    # Nominal qubit parameters for the inherited ConcentricTransmon
    'r_inner': 110,
    ...
}
```

Create a Simulation script

In a separate file importing the class (or in the same file for brevity)

First, let's store our arguments for the Simulation class in the `sim_parameters` dictionary, such that, we will eventually call `sim_class(…, **sim_parameters)`. This will be useful in a moment…

```
elmer_export_parameters = {
    'path': dir_path,
    'tool': 'capacitance',
    'workflow': {
        'run_gmsh_gui': False,
        'run_elmergrid': True,
        'run_elmer': True,
        'run_paraview': False
        'n_workers': 1
        'gmsh_n_threads': 8,
        'elmer_n_processes': 8,
        'python_executable': 'python'
    },
    'linear_system_method': 'mg',
    'p_element_order': 2,
    'mesh_size': {
        'global_max': 50.,
        'gap&signal': [2., 8.],
        'gap&ground': [2., 8.],
        'port': [1., 8.],
    }
}
```

Setting Elmer parameters

Details for the settings are in the export_elmer API and can change in the near future as Elmer development accelerates.

Number of processes and threads to use is set here along with the meshing parameters. Elmer does not currently use adaptive meshing so the mesh should be as fine as you need.

Parameter sweeps

```python
# Get layout
logging.basicConfig(level=logging.WARN, stream=sys.stdout)
layout = get_active_or_new_layout()
simulations = [sim_class(layout, **sim_parameters)]

# Sweep given parameters independently
simulations += sweep_simulation(layout, sim_class, sim_parameters, {
    # The nominal `sim_parameters` are overwritten with these
    'r_inner': [70, 90, 100],
    'r_outer': np.linspace(270, 290, 3),
})

# Full ND sweep of given parameters
simulations += cross_sweep_simulation(layout, sim_class, sim_parameters, {
    'r_inner': [100, 110, 120],
    'r_outer': np.linspace(270, 290, 3),
})


# Export simulation files
export_elmer(simulations, **elmer_export_parameters)

# Write and open OAS file
open_with_klayout_or_default_application(export_simulation_oas(simulations, dir_path))
```

To sweep parameters in the class (qubit in this case), we may use dictionary inputs for

`sweep_simulation` or `cross_sweep_simulation`

to easily generate Simulation classes with independent and full-ND sweeps, respectively.

Of course, any for-loops etc. for adding simulations are valid as well.

You should now be able to run the Python script
and the resulting `simulation.sh` as was done in the Elmer section!

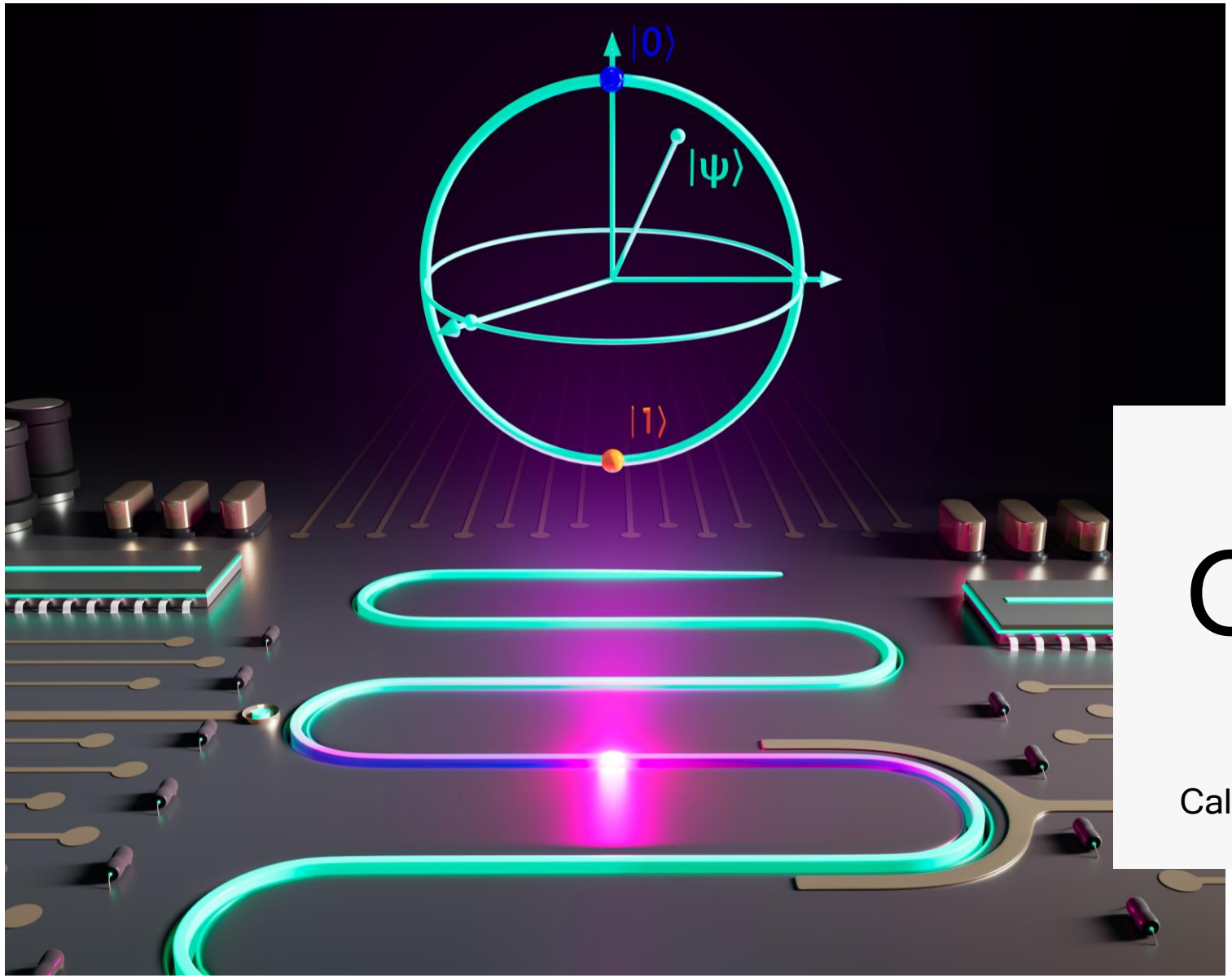


Figure. Artistic impression of a unimon qubit in a quantum processor. Credits: Aleksandr Kakinen.

Qubit analysis

Calculating qubit properties from the C-matrix

Compute C_Σ

After running some simulations for our qubit, we should look at the resulting properties.

For a floating transmon, the properties are determined by the junction energy

$$E_\mathrm{J} = \frac{1}{hL_\mathrm{J}}\left(\frac{\Phi_0}{2\pi}\right)^2$$

and the charging energy

$$E_\mathrm{C} = \frac{e^2}{2hC_\Sigma}$$

Let's now compute C_Σ and presume some junction inductance from literature L_J.

Subsequently, we use the open-source scQubits library from Jens Koch's group for getting the Hamiltonian.

```
# THIS CODE IS ALSO PROVIDED IN A NOTEBOOK
import json
from pathlib import Path
import numpy as np
from matplotlib import pyplot as plt
import scqubits as scq

result_jsons = Path('results/').rglob('*results.json')

# Let's look at one result
result_json = next(result_jsons)
with open(result_json) as file_pointer:
    result = json.load(file_pointer)

# For this case we should add a capacitive contribution of a full-length resonator to C_33.
# For 50 ohm and f=6 GHz,, this is roughly something like 450 fF.
# This will affect the C_Sigma little but will be relevant for estimating coupling g.

result['CMatrix'][2][2] += 450e-15

def C_Sigma_two_islands(data):
    r""" Data argument is a dict with `CMatrix` key of a 3x3 capacitance matrix with a coupler
as the last port.
    Derived with the Lagrangian. See the following references for similar derivations:

    [1] F. Marxer et al., “Long-distance transmon coupler with CZ gate fidelity above 99.8%”.
arXiv:2208.09460, Dec. 19, 2022.
    [2] A. P. A. Cronheim, “A Circuit Lagrangian Formulation of Opto-mechanical Coupling
between two Electrical Resonators mediated by a SQUID”.
        Delft University of Technology, Dec. 10, 2018. [Online]. Available:
http://resolver.tudelft.nl/uuid:a4c72663-65c9-4857-8ffa-ebaf2cbc9782

    Returns:
        C_Sigma: Effective qubit total capacitance for :math:`C_\Sigma`.
        C_q: Qubit island-island capacitance with correction from coupler.
        C_r: Resonator capacitance with correction from qubit islands.
        C_qr: Effective coupling capacitance between the qubit and the resonator.
    """
    C_sim = data['CMatrix']
    C_theta = (C_sim[0][0] + C_sim[0][2] + C_sim[1][1] + C_sim[1][2])
    C_q = C_sim[0][1] + ((C_sim[0][0] + C_sim[0][2]) * (C_sim[1][1] + C_sim[1][2])) / C_theta
    C_r = C_sim[0][2] + C_sim[1][2] + C_sim[2][2] - (C_sim[0][2] + C_sim[1][2]) ** 2 / C_theta
    C_qr = (C_sim[0][2] * C_sim[1][1] - C_sim[1][2] * C_sim[0][0]) / C_theta
    return C_q - C_qr ** 2 / C_r, C_q, C_r, C_qr

C_Sigma, C_q, C_r, C_qr = C_Sigma_two_islands(result)
print(f'{C_Sigma * 1e15} fF')
```

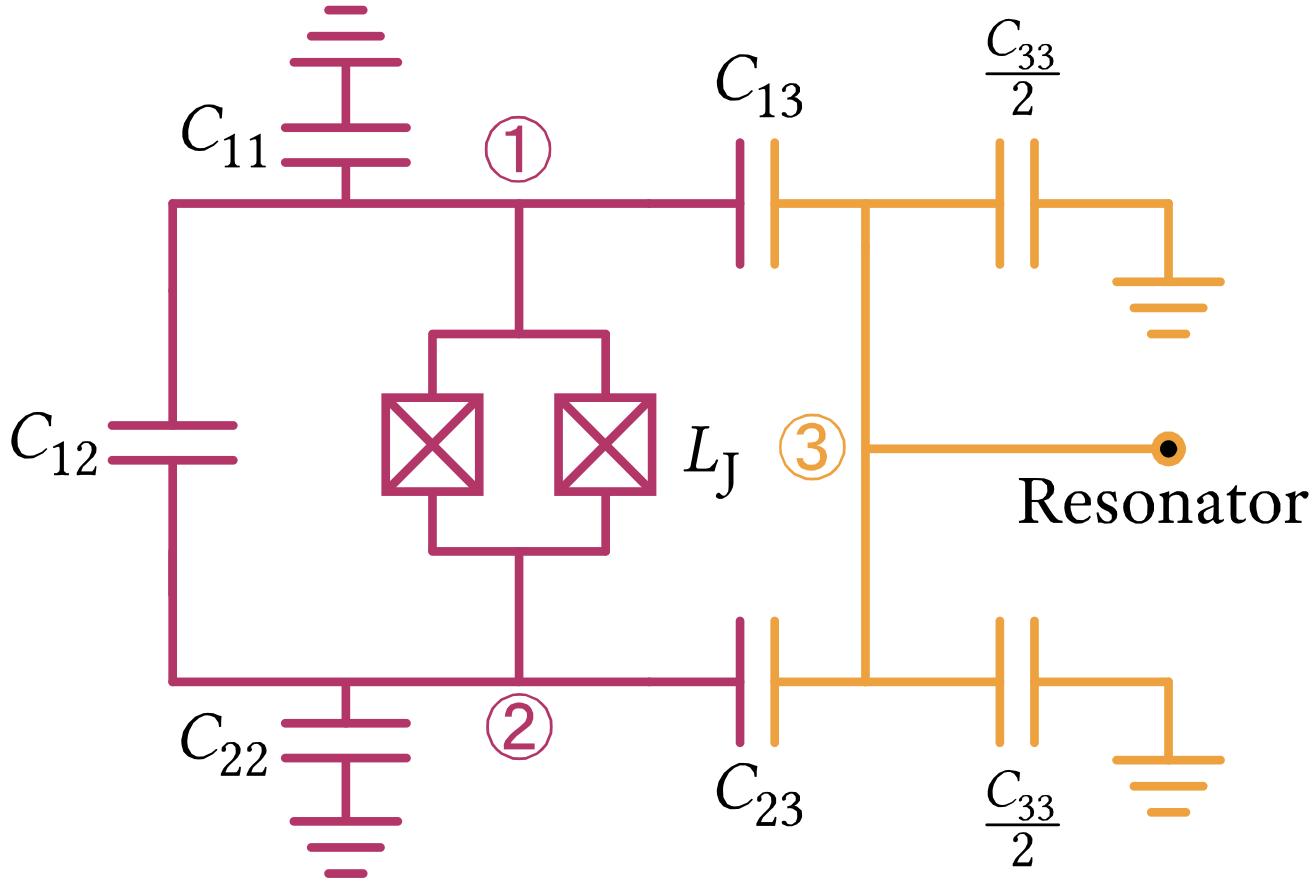
Lumped-element model for C_Σ

$$C_\mathrm{q} = C_{12} + \frac{(C_{11}+C_{13})(C_{22}+C_{23})}{C_{11}+C_{13}+C_{22}+C_{23}}$$

$$C_\mathrm{r} = C_{13} + C_{23} + C_{33} - \frac{(C_{13}+C_{23})^2}{C_{11}+C_{13}+C_{22}+C_{23}}$$

$$C_\mathrm{qr} = \frac{C_{13}C_{22} - C_{23}C_{11}}{C_{11}+C_{13}+C_{22}+C_{23}}$$

$$C_\Sigma = C_\mathrm{q} - \frac{C_\mathrm{qr}^2}{C_\mathrm{r}}$$

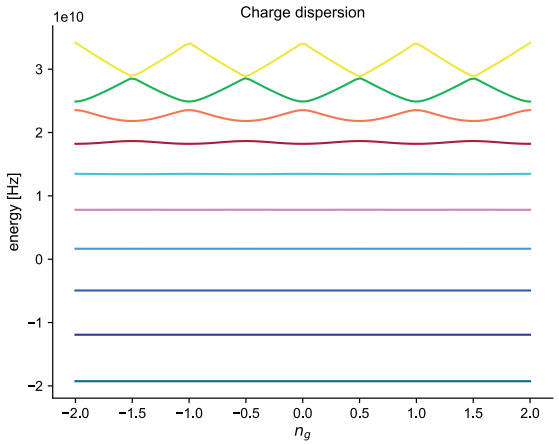


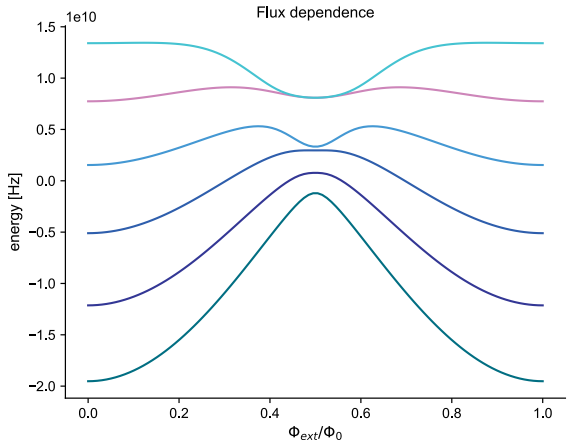
Energy vs. parameter

It’s easy to build your own design workflow around scQubits with qutip, skrf etc.

We can see how the transmon is well-protected against charge noise.

For more possibilities, consult the scQubits examples.






```
from scipy.constants import e, h, pi, physical_constants

L_J = 7.015e-9  # Example from E. Hyyppä et al., 'Unimon qubit', Nat Commun, vol. 13, no. 1, Art. no. 1, Nov. 2022

# Units on Hz for energy (E_J / h)
E_J = (physical_constants['mag. flux quantum'][0] / (2*pi)) ** 2 / (h * L_J)
E_C = e ** 2 / (2 * h * C_Sigma)
print(f'ratio = {E_J / E_C}')

scq.set_units('Hz')  # Energy is now given in Hertz
transmon = scq.TunableTransmon(
    EJmax=E_J,
    EC=E_C,
    d=0.1,
    flux=0.95,
    ng=0.5,  # Default charge dispersion
    ncut=30,
)

transmon.plot_wavefunction(which=(0, 1, 2, 3), mode='abs_sqr');

evals = transmon.eigenvals(3)
f_ge = evals[1] - evals[0]
print(f'f = {f_ge/1e9} GHz')

transmon.plot_evals_vs_paramvals(param_name='flux', param_vals=np.linspace(0, 1, 100));
plt.title('Flux dependence')

fig, axes = transmon.plot_evals_vs_paramvals('ng', np.linspace(-2, 2, 100), evals_count=10, subtract_ground=False)
plt.title('Charge dispersion')
```

Exercise: Coupling to a resonator

scQubits also supports composite Hilbert spaces. These can be used to model multiple qubits and couplers etc. with qutip support. See Composite Hilbert Spaces for details.

1. Your task is to implement the missing parts to get the anharmonicity and dispersive shift when connected to a 6 GHz resonator.

The coupling is given by

$$g \approx \frac{1}{2} \frac{C_{\mathrm{qr}}}{\sqrt{C_{\Sigma}\left(C_{\mathrm{r}} - \frac{C_{\mathrm{qr}}^2}{C_{\mathrm{q}}}\right)}} \sqrt{\omega_{\mathrm{q}} \omega_{\mathrm{r}}}$$

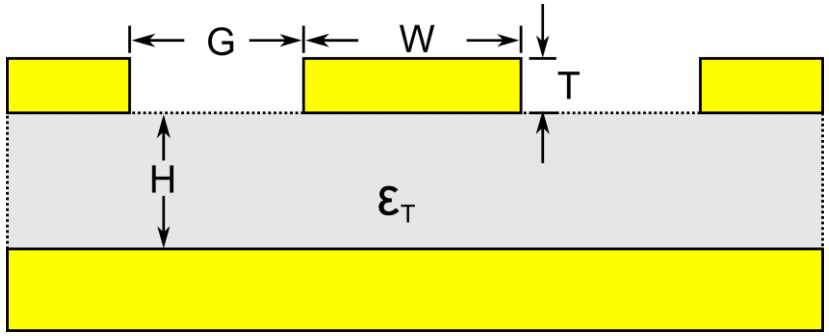
```
g = ...

resonator = scq.Oscillator(
    E_osc=...,
    truncated_dim=10
)

hilbert_space = scq.HilbertSpace([transmon, resonator])
# For details, see Eq. 3.3 in J. Koch et al., 'Charge-insensitive qubit design derived from
the Cooper pair box',
#   Phys. Rev. A, vol. 76, no. 4, p. 042319, Oct. 2007
hilbert_space.add_interaction(
    g_strength=g,
    op1=transmon.n_operator,
    op2=resonator.creation_operator,
    add_hc=True
)

eigenvalues = hilbert_space.eigenvals(evals_count=20)

hilbert_space.generate_lookup()
g0 = eigenvalues[0]  # lowest state
e0 = eigenvalues[hilbert_space.dressed_index((1, 0))]  # qubit is excited, resonator is ground
f0 = eigenvalues[hilbert_space.dressed_index((2, 0))]
g1 = eigenvalues[hilbert_space.dressed_index((0, 1))]
e1 = eigenvalues[hilbert_space.dressed_index((1, 1))]

f_ge_coupled = ...
f_ef_coupled = ...
f_rr_coupled_g = ...
f_rr_coupled_e = e1 - e0

print(f'anharmonicity = {(f_ef_coupled - f_ge_coupled) / 1e6} MHz', f'chi = {0.5 *
(f_rr_coupled_e - f_rr_coupled_g) / 1e6} MHz')
hilbert_space.hamiltonian()
```

Exercise: Coupling to a resonator

2. Instead of simply adding 450 fF to `C[2][2]`, add the correct value for a Z_0=100 Ω λ/4 CPW resonator with a frequency of 6 GHz. Does the qubit frequency change? How about the coupling?

You may use any online CPW calculators to get the properties. For a more programmable approach, something like scikit-rf may be used.

Use any reasonable values for the substrate properties.



`hilbert_space.hamiltonian()`

$$\mathcal{H} = \begin{pmatrix} -1.926\times10^{+10} & 5.000\times10^{+07} & 0.0 & 0.0 & 0.0 & \cdots & 0.0 \\ 5.000\times10^{+07} & -1.276\times10^{+10} & 7.071\times10^{+07} & 0.0 & 0.0 & \cdots & 0.0 \\ 0.0 & 7.071\times10^{+07} & -6.262\times10^{+09} & 8.660\times10^{+07} & 0.0 & \cdots & 0.0 \\ 0.0 & 0.0 & 8.660\times10^{+07} & 2.383\times10^{+08} & 1.000\times10^{+08} & \cdots & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.000\times10^{+08} & 6.738\times10^{+09} & \cdots & 4.989\times10^{+05} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.0 & 0.0 & 0.0 & 0.0 & 4.989\times10^{+05} & \cdots & 4.591\times10^{+10} \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 1.225\times10^{+08} \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 \end{pmatrix}$$

Summary

What we learned

- Install and run simulations with Elmer
- Setup your own simulations using KQCircuits
- View simulation results, and use them for further analysis with scQubits

Please ask questions on the KQCircuits Discord or GitHub Discussions so that others with similar questions can see the answers

Thanks!

Niko Savola

Quantum Engineer Intern

niko@meetiqm.com

nikosavola

www.meetiqm.com