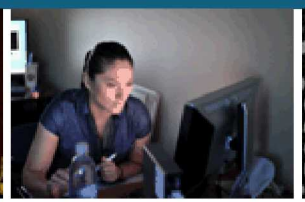
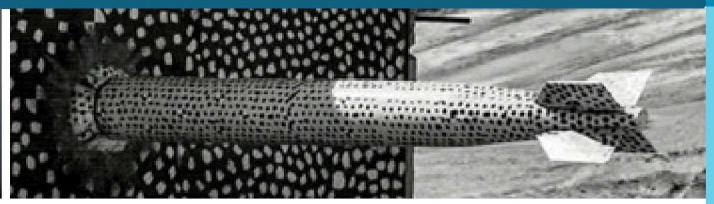
SAND2020-8327PE

Software@Sandia

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

PyGSTi

A Python implementation of Set Gate Tomography

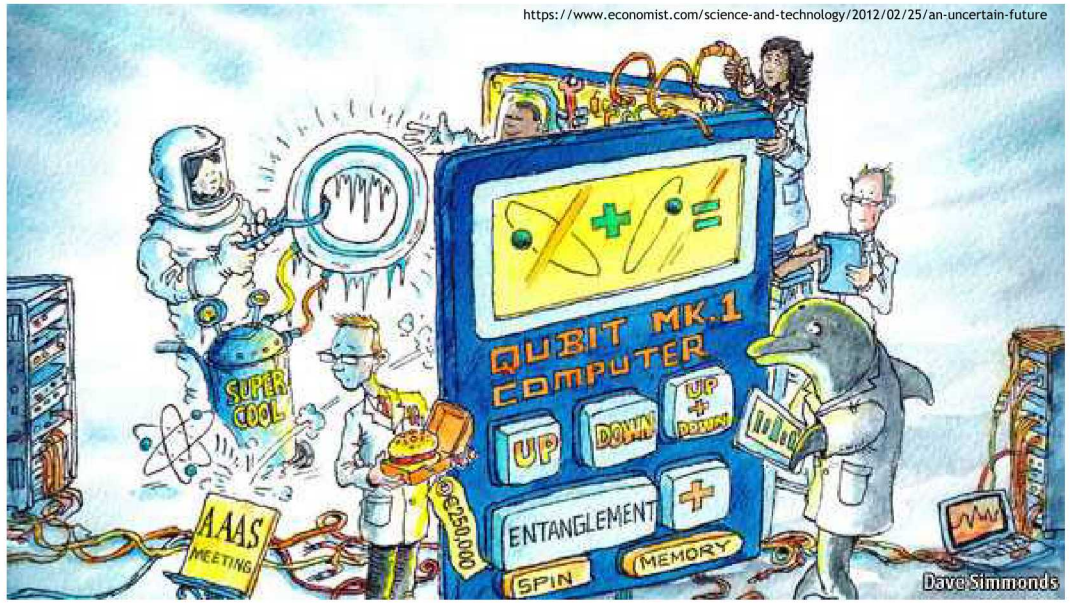
Erik Nielsen

Online Tutorial Series August, 2020

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Motivation: Quantum computers don't "just work"

Quantum computers are complex, finely tuned devices that try to preserve the coherence of a quantum state and control and measure that state. **While they have great promise** as a computing platform, **current quantum processors are small** (5-50 qubits) **and noisy** (have high error rates).



If today's quantum processor were a car…

To advance the field of quantum computing:

1. Running algorithms on existing systems can yield algorithmic insights. But this is confounded by unexpected failures. **An ability to accurately assess the capabilities of a quantum processor is essential** to employing it for useful research.
2. Larger systems with less noise must be built. **The better noise is understood in current devices, the better that noise can be combated in future versions**.

What does pyGSTi do?

The **pyGSTi** software **helps users quantify and understand what's not right with a quantum computer.**

It interfaces with a quantum computer at a high "computer science" level by running circuits/programs on the computer and analyzing the (classical) data that results. PyGSTi is essentially a diagnostic tool, and performs:

Automotive analog

- **Benchmarking**: Assess the **overall performance** of a quantum processor.
  - Clifford (standard), direct, and mirror randomized benchmarking (RB)
  - Volumetric benchmarking
  - Parity benchmarking

- **Model-based characterization**: Create a **detailed error model** capable of predicting the observed processor output. Insights into specific error mechanisms can ideally be gained by looking at the model.
  - Gate set tomography (the "GST" in pyGSTi)
  - Reduced-model tomography
  - Idle tomography

Broadly referred to as "**quantum characterization, validation, and verification**" (**QCVV**).

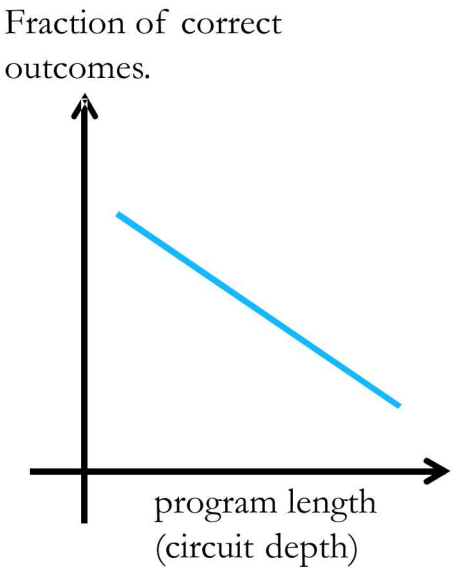
Benchmarking

Task: To provide a **holistic summary** of a quantum processor's **overall performance**

Solutions: Multiple protocols have been developed and implemented to address this task, but all follow the same basic paradigm, which is to:

1. Execute a collection of random programs (circuits) on the quantum computer for which you know the correct output. The circuits all come from a common "family" but have various depths (program lengths).

2. The fraction of correct outcomes (ideally 100%) for each circuit, along with it's depth information, are fit with a simple decay curve to arrive at an overall estimate of the error-per-depth present in the processor (for the specific circuit family that was used).



The **technically challenging** pieces of benchmarking algorithms are:

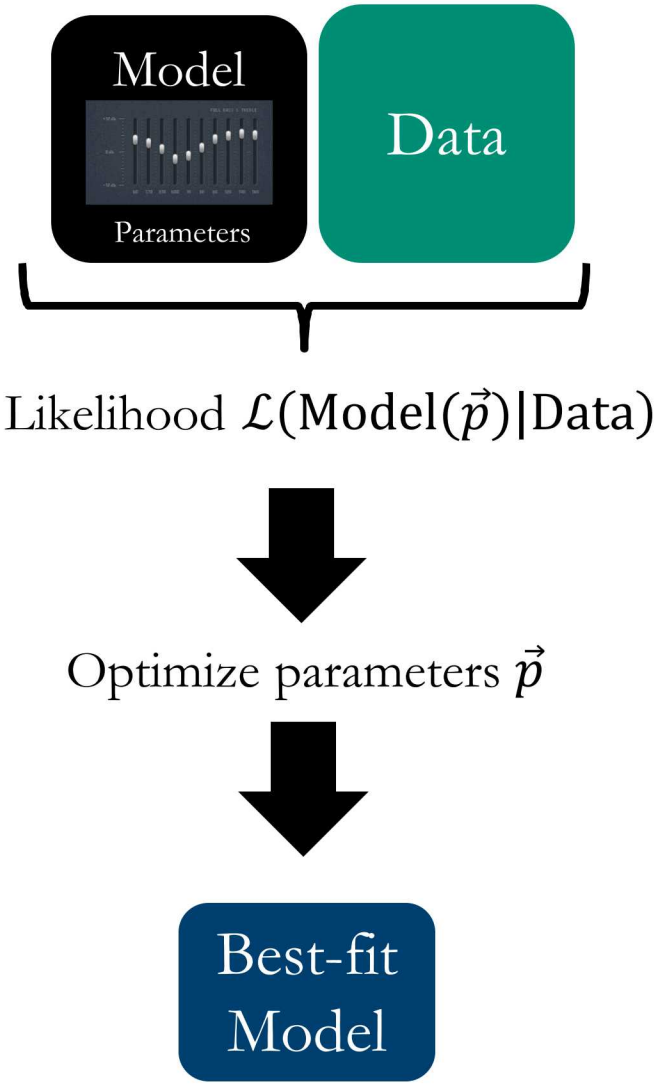
- **Choosing a circuit family** (result should be meaningful; circuits must have known oucome)
- **Sampling from a distribution** (that may be difficult to sample from, e.g. 2-qubit Cliffords).

Model-based characterization

Task: To provide a **detailed model** of a quantum processor's **per-circuit performance**

Solutions: Gate set tomography (GST) and variants thereof form the core protocols that address this task. The basic GST algorithm is to:

1. Select a class of parameterized models that constitutes your "search space".
2. Execute a collection of highly structured programs (circuits) that are sensitive to all the model parameters.
3. Optimize likelihood of the model given the data. This involves repeatedly simulating the circuits using the model located a particular parameter-space point and updating that point. The result is a best-fit model.

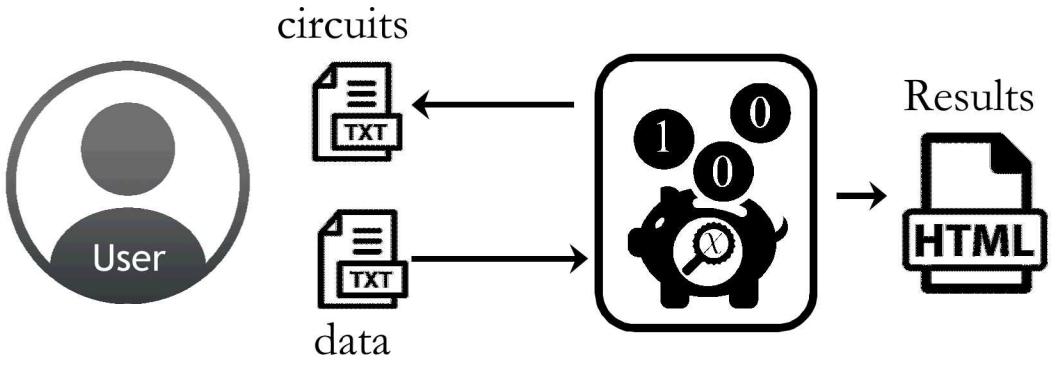


The **technically challenging pieces** of model-based characterization algorithms are:

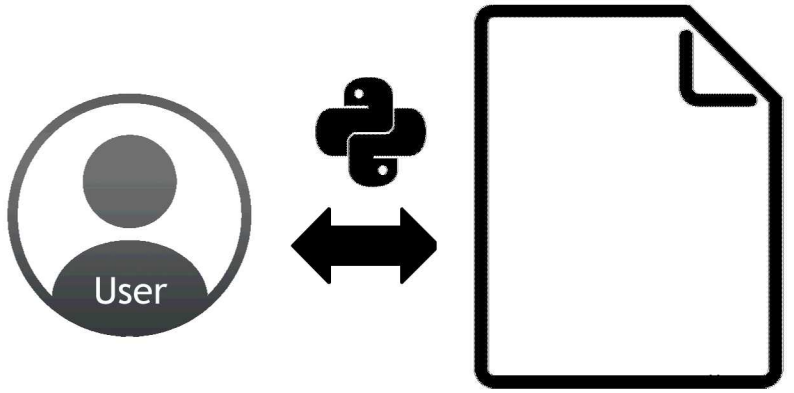
- **Choosing meaningful and efficient models** (describe data but not too many parameters)
- **Constructing structured circuits** that are sensitive to model parameters.
- **Simulating circuits** (exponential in # of qubits)
- **Optimization** over model space (a non-convex, many-parameter optimization)

Different ways of integrating pyGSTi into your workflow

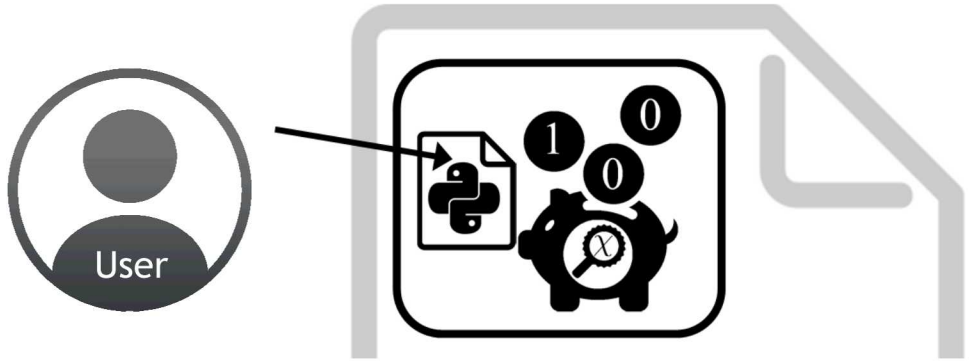
1. **As a standalone QCVV tool**:
   - pyGSTi acts as a separate application.
   - Interaction with other software is minimal, often using text files for I/O.
   - Often used for heavyweight protocols (e.g. GST).



2. **Within a Python analysis toolchain.**
   - pyGSTi functionality called as a subroutine
   - Useful for lightweight protocols (e.g. Model testing, RPE)



1. **Customized QCVV protocols.**
   - User-written code adds specific functionality to pyGSTi
   - pyGSTi calls user code as a subroutine.
   - Example: "Physical-model GST"



PyGSTi's `Protocol` pattern:

All of the protocols (benchmarking and characterization) are run using the same pattern within pyGSTi:

1. Choose a protocol; create* a `Protocol` object:

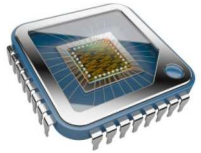
`Protocol`

2. Generate a collection of circuits compatible with that protocol – an *experiment design*.

`ExperimentDesign`

`Circuit`

3. Execute the circuits in the experiment design on your quantum computer. Save the results as a *data set*, which gets packaged with the experiment design into a `ProtocolData` object.



`ProtocolData`

`E.Design` `DataSet`

4. Run the protocol on the data:

`Protocol Results` = `Protocol` **.run(** `ProtocolData` **)**

5. Look at the results!

`ProtocolResults`

`Protocol`

`E.Design` `DataSet`

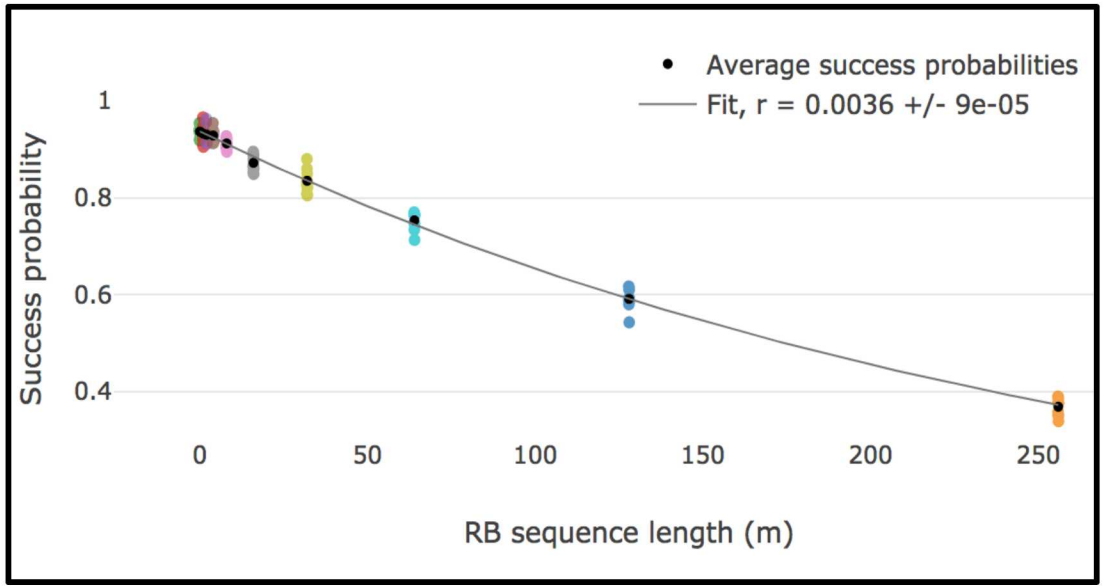
*can actually be deferred to step 4

Example: Randomized Benchmarking

Using pyGSTi to perform direct randomized benchmarking on 4 qubits:

```
protocol = pygsti.protocols.RB()  #Step 1
design = pygsti.protocols.DirectRBDesign(pspec,
                                         depths=[0, 1, 2, 4, 8, 16, 32, 64, 128, 256],
                                         circuits_per_depth=10,
                                         qubit_labels=['Q0','Q1','Q2', 'Q3']) #Step 2
pygsti.io.write_empty_protocol_data(design, 'example_rb', clobber_ok=True)
# --- Step3: TAKE DATA HERE (fill in dataset.txt) ---
data = pygsti.io.load_data_from_dir('example_rb')
results = protocol.run(data)  #Step 4

ws = pygsti.report.Workspace()
ws.init_notebook_mode(autodisplay=True)
ws.RandomizedBenchmarkingPlot(results)  #Step 5 ➔
```



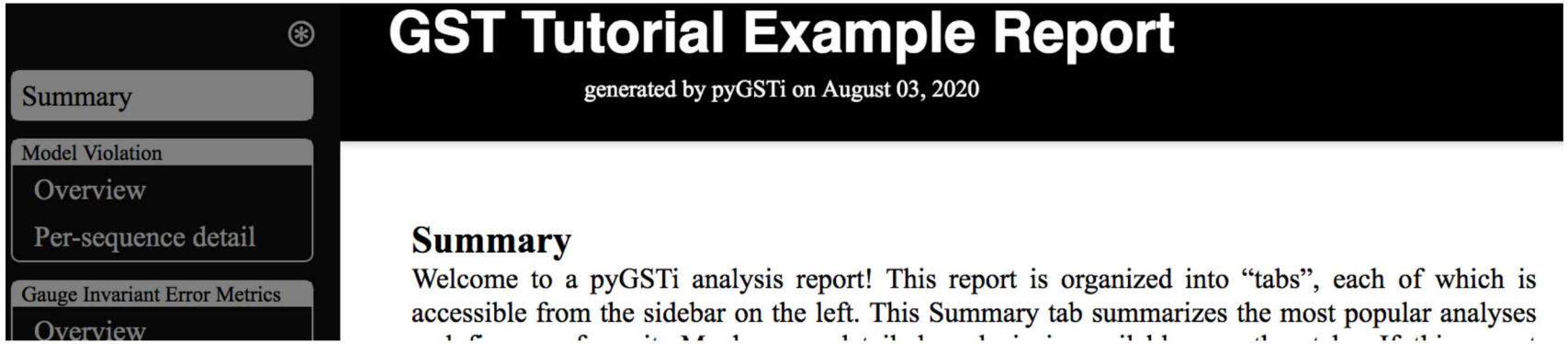
Example: Gate Set Tomography

Using pyGSTi to perform gate set tomography on 1 qubit:

```
protocol = pygsti.protocols.StandardGST('TP,CPTP,Target')  #Step 1
design = pygsti.protocols.StandardGSTDesign(smq1Q_XYI.target_model(),
         smq1Q_XYI.prep_fiducials(), smq1Q_XYI.meas_fiducials(),
         smq1Q_XYI.germs(), max_lengths=[1,2,4,8,16,32])  #Step 2

pygsti.io.write_empty_protocol_data(design, 'example_gst', clobber_ok=True)
# --- Step3: TAKE DATA HERE (fill in dataset.txt) ---
data = pygsti.io.load_data_from_dir('example_gst')
results = protocol.run(data)  #Step 4

report = pygsti.report.construct_standard_report(
    results, title="GST Tutorial Example Report", verbosity=2)  #Step 5
report.write_html("GSTExampleReport.html", single_file=True, verbosity=2)
```

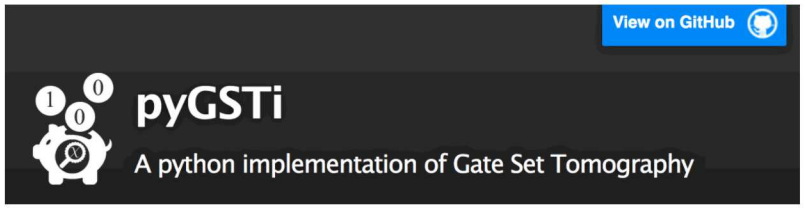


Open Research Problems

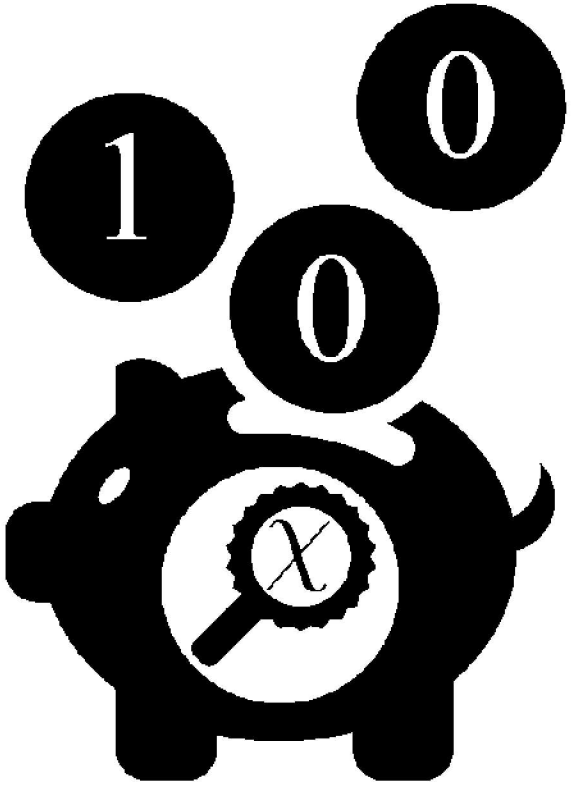
- Scaling model-based methods to larger number of qubits:
  - Circuit simulation cost is bottleneck
- Model-based methods on larger-than-gate-level components of a processor.
  - E.g., our recently developed "parity benchmarking" performs benchmarking on a parity-check circuit.
- Development of random or structured circuits to benchmark large (10-100 qubit) quantum processors.
  - Circuits must have known expected outcomes and also probe the processor well.
- Real-time methods for tune-up and/or characterization:
  - Almost all QCVV methods execute many circuits and then analyze the resulting data in bulk. Protocols that build and retain knowledge of the characterized state of a processor are interesting.

Technical information about pyGSTi

- Written in Python.
  - Now requires Python 3.5+
  - Around 150K lines of code (medium size?)
  - Contains optional compiled C extensions (can function as a pure Python package)
  - GitHub www.pygsti.info



- Software dependencies:
  - numpy, scipy, plotly
- Continuous integration tests
  - Linting
  - Unit tests
  - System tests
- Documentation
  - Tutorials
  - Online documentation at pygsti.readthedocs.io
- Contributions & feedback:
  - We welcome feedback and input from outside users.
  - Non-Sandia-employees must sign a contributor license agreement (CLA) in order for to make contributions.
- License: Apache 2.0



Publications

Publications primarily about pyGSTi:

- Erik Nielsen *et al,* "Probing quantum processor performance with `pyGSTi`" 2020 *Quantum Sci. Technol.* **5** 044002

Publications using pyGSTi to perform experimental research:

- R. Blume-Kohout, J. K. Gamble, E. Nielsen, K. Rudinger, J. Mizrahi, K. Fortier, and P. Maunz, Nat. Commun. 8, 14485 (2017).
- J. P. Dehollain, J. T. Muhonen, R. Blume-Kohout, K. M. Rudinger, J. K. Gamble, E. Nielsen, A. Laucht, S. Sim- mons, R. Kalra, A. S. Dzurak, et al., New J. Phys. 18, 103018 (2016).
- T. J. Proctor, A. Carignan-Dugas, K. Rudinger, E. Nielsen, R. Blume-Kohout, and K. Young, Phys. Rev. Lett. 123, 030503 (2019).
- D. Kim, D. Ward, C. Simmons, J. K. Gamble, R. Blume- Kohout, E. Nielsen, D. Savage, M. Lagally, M. Friesen, S. Coppersmith, et al., Nat. Nanotechnol. 10, 243 (2015).
- K. Rudinger, S. Kimmel, D. Lobser, and P. Maunz, Phys. Rev. Lett. 118, 190502 (2017).
- M. Rol, C. Bultink, T. O'Brien, S. de Jong, L. Theis,
- X. Fu, F. Luthi, R. Vermeulen, J. de Sterke, A. Bruno, et al., Phys. Rev. Appl. 7, 041001 (2017)
- K. Rudinger, T. Proctor, D. Langharst, M. Sarovar, K. Young, and R. Blume-Kohout, Phys. Rev. X 9, 021045 (2019).
- S. Mavadia, C. L. Edmunds, C. Hempel, H. Ball, F. Roy, T. M. Stace, and M. J. Biercuk, npj Quantum Informa- tion 4, 7 (2018).
- M. Ware, G. Ribeill, D. Riste, C. A. Ryan, B. John- son, and M. P. da Silva, arXiv preprint arXiv:1803.01818 (2018).
- T. Proctor, M. Revelle, E. Nielsen, K. Rudinger, D. Lob- ser, P. Maunz, R. Blume-Kohout, and K. Young, arXiv preprint arXiv:1907.13608 (2019).
- G. A. L. White, C. D. Hill, and L. C. L. Hollenberg, "Performance optimisation for drift-robust fidelity im- provement of two-qubit gates," (2019), arXiv:1911.12096 [quant-ph].
- Y. Chen, M. Farahzad, S. Yoo, and T.-C. Wei, Phys. Rev. A 100, 052315 (2019).
- M. Sarovar, T. Proctor, K. Rudinger, K. Young, E. Nielsen, and R. Blume-Kohout, "Detecting crosstalk errors in quantum information processors," (2019), arXiv:1908.09855 [quant-ph].
- T. L. Scholten, Y.-K. Liu, K. Young, and R. Blume- Kohout, "Classifying single-qubit noise using machine learning," (2019), arXiv:1908.11762 [quant-ph].
- L. C. G. Govia, G. J. Ribeill, D. Rist`e, M. Ware, and H. Krovi, "Bootstrapping quantum process tomography via a perturbative ansatz," (2019), arXiv:1902.10821 [quant-ph].
- S. S. Hong, A. T. Papageorge, P. Sivarajah, G. Crossman, N. Didier, A. M. Polloreno, E. A. Sete, S. W. Turkowski, M. P. da Silva, and B. R. Johnson, Phys. Rev. A 101, 012302 (2020).
- M. R. Geller, "Rigorous measurement error correction," (2020), arXiv:2002.01471 [quant-ph].
- M. K. Joshi, A. Elben, B. Vermersch, T. Brydges, C. Maier, P. Zoller, R. Blatt, and C. F. Roos, "Quan- tum information scrambling in a trapped-ion quantum simulator with tunable range interactions," (2020), arXiv:2001.02176 [quant-ph].

The future of pyGSTi

- PyGSTi continues to be under active development, funded under multiple projects at Sandia National Labs.
- While PyGSTi continues to implement **new protocols and features**, specifically focusing on characterization of larger systems, there is also effort to **mature the codebase**. In particular, reaching version 1.0, marked by more stringent testing and a more stable API, are near-term goals.
- There is a desire to integrate more with other existing tools (particularly in the area of circuit simulation).

Software Points of Contact

- Relvant URLs:
  - pyGSTi on GitHub: www.pygsti.info
  - Documentation: pygsti.readthedocs.io
  - Quantum Performance Lab: qpl.sandia.gov
- Developers:
  - Erik Nielsen
  - Timothy Proctor
  - Kenneth Rudinger
  - Antonio Russo
  - Kevin Young
  - Robin Blume-Kohout
  - Robert Kelly
- Direct questions and comments to: pygsti@sandia.gov

group photo

Extra content: full scripts for running examples (including data generation)

RB Example

```
import pygsti

#Step 0: create a processor specificiation (defines your device)
# n_qubits = 4
# qubit_labels = ['Q0','Q1','Q2','Q3']
# gate_names = ['Gxpi2', 'Gxmpi2', 'Gypi2', 'Gympi2', 'Gcphase']
# availability = {'Gcphase':[('Q0','Q1'), ('Q1','Q2'), ('Q2','Q3'), ('Q3','Q0')]}
# pspec = pygsti.obj.ProcessorSpec(n_qubits, gate_names, availability=availability,
qubit_labels=qubit_labels, construct_models=('clifford',))
# pickle.dump(pspec, open('MyProcessorSpec.pkl','wb'))

import pickle
pspec = pickle.load(open('MyProcessorSpec.pkl','rb'))

# Step 1: choose a protocol
protocol = pygsti.protocols.RB()

#Step 2: Create an experiment design for doing Direct RB on our 4-qubit processor
design = pygsti.protocols.DirectRBDesign(pspec,
                                depths=[0, 1, 2, 4, 8, 16, 32, 64, 128, 256],
                                circuits_per_depth=10,
                                qubit_labels=['Q0','Q1','Q2', 'Q3'],
                                sampler='edgegrab',
                                samplerargs=[0.5],
                                randomizeout=True,
                                citerations=20)
#Step 3: Take data
pygsti.io.write_empty_protocol_data(design, 'example_rb', clobber_ok=True)
# **STOP** - take data here by filling in the `dataset.txt` file produced in the last line.
data = pygsti.io.load_data_from_dir('example_rb')

#Step4: Run the protocol
results = protocol.run(data)

#Step5: Look at the results
ws = pygsti.report.Workspace()
ws.init_notebook_mode(autodisplay=True)
ws.RandomizedBenchmarkingPlot(results)

gate_error_rate = 0.001; n_qubits = 4
print("The error rate we approximately expect accord to Direct RB theory = ", 1 - (1 -
gate_error_rate)**n_qubits))

# Construct an error model with 0.1% local depolarization on each qubit after each gate.
gate_error_rate = 0.001
data_template_filename = 'example_rb/data/dataset.txt'

qubit_labels = ['Q0','Q1','Q2','Q3']
gate_names = ['Gxpi2', 'Gxmpi2', 'Gypi2', 'Gympi2', 'Gcphase']
availability = {'Gcphase':[('Q0','Q1'), ('Q1','Q2'), ('Q2','Q3'), ('Q3','Q0')]}
pspec = pygsti.obj.ProcessorSpec(len(qubit_labels), gate_names, availability=availability,
                        qubit_labels=qubit_labels, construct_models=('TP',))
noisemodel = pspec.models['TP'].copy()
for gate in noisemodel.operation_blks['gates'].values():
   if gate.dim == 16:
      gate.depolarize(1 - pygsti.tools.rbtools.r_to_p(1 - (1-gate_error_rate)**2, 4))
   if gate.dim == 4:
      gate.depolarize(1 - pygsti.tools.rbtools.r_to_p(gate_error_rate, 2))
pygsti.io.fill_in_empty_dataset_with_fake_data(noisemodel, data_template_filename,
num_samples=1000, seed=1234)
```

Extra content: full scripts for running examples (including data generation)

GST Example

```
import pygsti
#Step 0: import a standard "model-pack", essentially a 1-qubit processor spec & more for our device
from pygsti.modelpacks import smq1Q_XYI

#Step 1: choose a protocol
protocol = pygsti.protocols.StandardGST('TP,CPTP,Target')

#Step 2: create an "experiment design" for doing GST on the std1Q_XYI gate set
design = pygsti.protocols.StandardGSTDesign(smq1Q_XYI.target_model(), smq1Q_XYI.prep_fiducials(), smq1Q_XYI.meas_fiducials(),
                                smq1Q_XYI.germs(), max_lengths=[1,2,4,8,16,32])

#Step 3: Take data
pygsti.io.write_empty_protocol_data(design, 'example_gst', clobber_ok=True)
# **STOP** - take data here by filling in the `dataset.txt` file produced in the last line.
data = pygsti.io.load_data_from_dir('example_gst')

#Step 4: Run the protocol
results = protocol.run(data)

#Step 5: Look at the data (generate a HTML report)
report = pygsti.report.construct_standard_report(
   results, title="GST Tutorial Example Report", verbosity=2)
report.write_html("GSTExampleReport.html", single_file=True, verbosity=2)

# Look at the report [here](GSTExampleReport.html)


#Generate GST Data
import pygsti
from pygsti.modelpacks import smq1Q_XYI
data_template_filename = 'example_gst/data/dataset.txt'
datagen_model = smq1Q_XYI.target_model().depolarize(op_noise=0.01, spam_noise=0.001)
pygsti.io.fill_in_empty_dataset_with_fake_data(datagen_model, data_template_filename, num_samples=1000, seed=1234)
```