Python is the Ultimate MATLAB Toolbox

Albert Danial

Dec. 17, 2022

Contents

Chapter 1

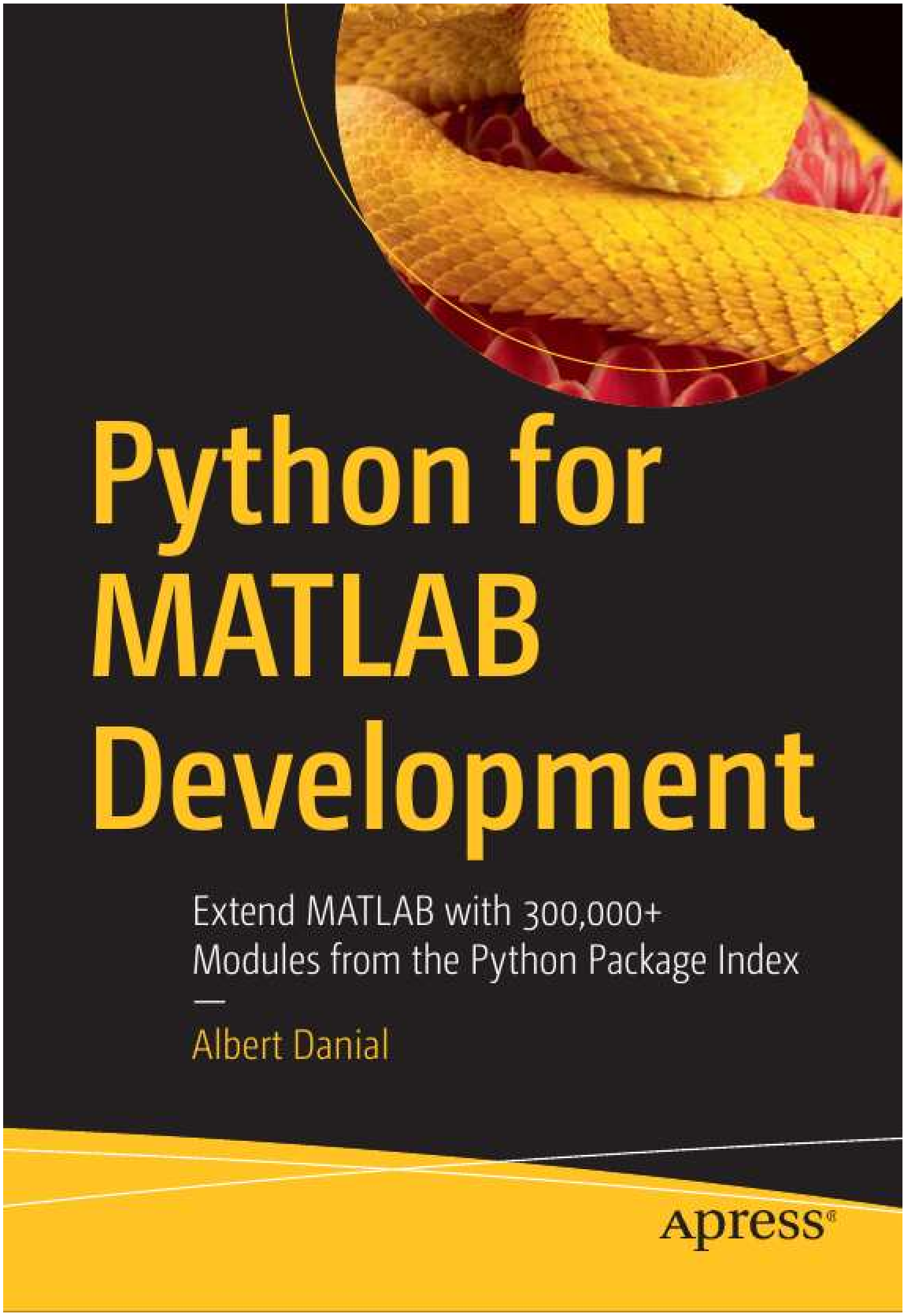
Introduction

This ebook is a compilation of articles posted to
https://al.danial.org/posts/python_is_the_ultimate_matlab_toolbox/
between May and November, 2022. The articles themselves are excerpts and variations of examples from my book, *Python for MATLAB Development* (PfMD). PfMD shows you how to enhance MATLAB with Python solutions to a vast array of computational problems in science, engineering, statistics, finance, and simulation. It is three books in one:

- A thorough Python tutorial that leverages your existing MATLAB knowledge with a comprehensive collection of MATLAB/Python equivalent expressions
- A reference guide to setting up and managing a Python environment that integrates cleanly with MATLAB
- A collection of recipes that demonstrate Python solutions invoked directly from MATLAB

The demonstrated techniques and explanations will help you solve your own challenging problems in MATLAB using open source Python modules.

The book you're reading now, *Python is the Ultimate MATLAB Toolbox*, is freely available (Creative Commons license) in PDF and ePub formats from
`https://github.com/AlDanial/matlab_with_python/ultimate_toolbox_book`. It is a brief tour of what Python can offer MATLAB developers.



Python for MATLAB Development

Extend MATLAB with 300,000+ Modules from the Python Package Index

—

Albert Danial

Apress®

Chapter 2

Python Is The Ultimate MATLAB Toolbox

2022/05/10

MATLAB's ability to run Python code is fantastic—it extends MATLAB's capabilities to encompass everything Python can do. Oddly, few MATLAB developers I know are aware of MATLAB's binary API to Python, or of the vast possibilities Python offers MATLAB developers. I wrote *Python for MATLAB Development* to teach Python to MATLAB programmers and to demonstrate many of the benefits Python can bring them. My presentation at the MATLAB EXPO 2022 (slides) shows the fundamentals of calling Python code from MATLAB.

Chapter 3 will help you set up a stable MATLAB+Python environment. The chapters after that show how Python helps MATLAB to:

- Read and write YAML, ini, TOML files with `pyyaml`, `configparser`, `toml` (*PfMD* § 7.2-7.5). Describes `py2mat.m`, `mat2py.m`, (*PfMD* § 6.5.7), `df2t.m` and `t2df.m` which greatly simplify MATLAB/Python data conversions.
- Read and write SQLite databases in MATLAB with Python shows how to create, query, and modify SQLite databases in MATLAB using Python's sqlite3 module. (*PfMD* § 7.12)
- Write richly-formatted Excel spreadsheets with openpyxl (*PfMD* § 7.8)
- Interact with PostgreSQL using the `psycopg2` Python module (*PfMD* § 7.20)
- Interact with MongoDB using the `pymongo` Python module (*PfMD* § 7.21)
- Set/get/subscribe to key-value data in Redis with `pyredis`(*PfMD* § 7.22)
- Solve global optimization problems with `scipy.optimize.differential_evolution()` (*PfMD* § 11.6)
- Solve simultaneous nonlinear equations with `scipy.optimize.root()` (*PfMD* § 11.11)

- Solve integrals, derivatives, Laplace transforms, series summations symbolically with SymPy (*PfMD* § 11.15, 11.16)
- Accelerate MATLAB with Python and Numba (*PfMD* § 14.10, 14.12)
- Distribute MATLAB Computations To Python Running on a Cluster with Dask `dask` (*PfMD* § 14.14)

2.1 What about toolboxes from the MathWorks?

The MathWorks has a large collection of toolboxes that can perform most of the tasks above. Their toolboxes are the preferred solutions because they offer

- **unique capability**: many MATLAB toolboxes solve such complex or specialized problems that no alternatives–Python or otherwise–exist
- **programming convenience**: a solution coded entirely in MATLAB is easier to write, debug, and maintain
- **configuration convenience**: you don't need to create a special MATLAB-friendly Python virtual environment or configure your MATLAB setup to use it
- **stability**: hybrid MATLAB/Python solutions are vulnerable to library compatibility problems caused by routine operating system updates. MathWorks toolboxes don't have this problem.

Python-based solutions become attractive if you don't have timely access to to the necessary MathWorks toolboxes. License procurement at large organizations often takes months and irritates both budget managers and the people needing the toolbox. Software purchase requests are typically challenged with questions like: Why was this need not anticipated? Which project's funds should we cut to pay for this? Can you justify raising our overhead rates with this purchase? Is this a one-time need or will you need it again next year? If one can get organizational approval and can wait a few months, then MathWorks toolboxes are the way to go.

2.2 What about the FileExchange?

The MathWorks' code sharing site, FileExchange, hosts more than 40,000 freely available packages to supplement MATLAB, including solutions to several of the Python-powered options listed above. Why turn to Python when pure-MATLAB options exist on the FileExchange? There are several reasons:

- Value-added Python distributions like Anaconda come bundled with a large collection of popular modules for numeric, scientific, engineering, and financial computation. If your organization supports Anaconda, chances are good that Python modules you need to augment your MATLAB work are already available.

- Organizations that have commercial support agreements with Python distributors like Anaconda and ActiveState, or Linux providers such as Red Hat, have access to security-vetted packages from these vendors' curated repositories. Installing Python modules from these sources is a safer bet than pulling files from the FileExchange.

- The Python community is several times larger than MATLAB's. This means popular Python modules (including all listed above) see heavier use and so tend to be more extensively exercised and feature-rich than corresponding FileExchange options. Although project star counts are crude proxies for popularity and use, compare the number of stars in two popular MySQL client interfaces: the MATLAB MySQL Database Connector has 32 stars on the FileExchange while Python's PyMySQL module (which itself is less popular than the official MySQL connector released by the MySQL development team) has 6,800 stars on Github.

- Open source Python modules tend to be developed, discussed, tracked, and released on code collaboration sites such as Github, Gitlab, or SourceForge. These allow one to report and track bugs, track forks, follow code commits, and view code in the browser. In contrast, the FileExchange is essentially a storage location with options for comments. Compared to the collaboration sites, the FileExchange offers less insight on a project's vitality, code content, or number of contributors.

Chapter 3

Set up a MATLAB+Python environment

2022/05/10

3.1 What you'll need

- MATLAB version 2020b or newer (some MATLAB versions older than 2020b support Python, but the farther back you go, the more more limitations and problems you'll hit). The latest release (2022b as of this writing) is definitely recommended.
- A Python installation supported by your version of MATLAB. I strongly recommend the Anaconda Python distribution [1] for two reasons: 1) it comes with a deep stack of Python modules and external tools useful for scientific computing and 2) its `conda` command has the ability to create virtual environments with specific shared library versions. Control over shared library versions in Python is invaluable for troubleshooting MATLAB errors (or crashes) when importing complex modules such as NumPy, SciPy, and statsmodels (among others) in MATLAB.

3.2 Step 1: `pyenv`

The first command to issue in a MATLAB console is `pyenv`. It will either come up empty, or it will print information about the Python installation MATLAB found:

```
>> pyenv
ans =
  PythonEnvironment with properties:

          Version: "3.9"
       Executable: "/usr/local/anaconda3/2021.05/envs/matpy/bin/python"
          Library: "/usr/local/anaconda3/2021.05/envs/matpy/lib/libpython3.9.so"
             Home: "/usr/local/anaconda3/2021.05/envs/matpy"
           Status: Loaded
    ExecutionMode: InProcess
        ProcessID: "840228"
```

If `pyenv` comes up empty, set `'Version'` to the path of the Python executable you want to use:

```
>> pyenv('Version', "/usr/local/anaconda3/2021.05/envs/matpy/bin/python")
```

On a Windows computer you entry might be

```
>> pyenv('Version', "C:/ProgramData/Anaconda3/python.exe")
```

(Yes, forward slashes work in Windows too.)

The best way to have MATLAB recognize a specific virtual environment is to first activate the virtual environment in a terminal, then start MATLAB from that terminal.

3.3 Step 2: Does it work?

A quick way to determine if your MATLAB and Python environments are compatible is to import the NumPy module into MATLAB:

```
>> np = py.importlib.import_module('numpy')
```

NumPy and MATLAB have several shared libraries in common so importing NumPy into MATLAB is a good litmus test for overall MATLAB/Python compatibility. Several things may happen:

1. A long list of function names appears.
2. `Python Error: ImportError`

3. `Unable to resolve the name numpy`
4. `Segmentation violation detected`

The first case indicates the import was successful. If you see the functions, congratulations, you should be able to use MATLAB and Python together effectively.

The last three errors can be caused by a Python installation that is not version-consistent with, or not visible to MATLAB; the Python installation does not have NumPy installed; or you need to create a virtual environment configured with shared library versions that are consistent with MATLAB's shared libraries.

Possible work-arounds to library conflicts:

1. Change the `ExecutionMode` setting to `OutOfProcess` ExecutionMode: InProcess

   ```
   >> pyenv("ExecutionMode","OutOfProcess")
   ```

2. Create a conda environment with library versions that more closely match MATLAB's. Unfortunately there's no obvious way to identify either the libraries or the versions so experimentation is needed. The conda environment that works well for me with MATLAB 2020b is

   Linux (Ubuntu 20.04) + MATLAB 2020b

   ```
   conda create --name matpy python=3.8.8 libgcc-devel_linux-64=8.4.0  \
       libgcc-ng=8.4.0 geopandas matplotlib pulp cartopy austin faker  \
       pint poliastro psycopg2 pymongo pythran redis redis-py simanneal \
       netCDF4 descartes h5py statsmodels pyyaml psutil lxml dask       \
       distributed paramiko sympy requests pyflakes uncertainties seaborn \
       cython pytest scikit-umfpack ipython -c conda-forge
   ```

   Windows 10 + MATLAB 2020b

   ```
   conda create --name matpy python=3.8.8 geopandas matplotlib pulp ^
       cartopy faker pint poliastro psycopg2 pymongo pythran redis  ^
       redis-py statsmodels simanneal netCDF4 descartes h5py pyyaml ^
       psutil lxml dask distributed paramiko sympy requests pyflakes ^
       cython uncertainties seaborn pytest ipython -c conda-forge
   ```

   macOS (Catalina, Big Sur) + MATLAB 2020b

```
conda create --name matpy python=3.8.8 geopandas matplotlib pulp \
    cartopy austin faker intel-openmp=2021.2.0 llvm-openmp=10.0.0 \
    libllvm10=10.0.1 llvmlite=0.36.0 \
    pint poliastro psycopg2 pymongo pythran redis redis-py simanneal \
    netCDF4 descartes h5py statsmodels pyyaml psutil lxml dask       \
    distributed paramiko sympy requests pyflakes uncertainties seaborn \
    cython pytest ipython -c conda-forge
```

I've found MATLAB 2022a and 2022b to work cleanly with the base Anaconda 2021.05 installation; no finely-tuned conda environment needed.

Chapter 4

Read YAML, ini, TOML files in MATLAB with Python

2022/05/27

4.1 YAML, TOML, ini: Convenient formats for program configuration data

As applications evolve, their inputs tend to become more complex. Parsing text files for program configuration data is a hassle so I typically store such data in formats such as YAML, TOML, `ini`, JSON, or XML. MATLAB natively supports JSON and XML but of the five listed options, these are the least attractive to me. JSON does not support comments and XML is just an all-around drag to view, edit, and code for. `ini` isn't great either but suffices for simple inputs.

FileExchange options exist for reading YAML, TOML, and `ini` files. Alternatively you can use Python to read and write files in these formats.

As an example, say you're writing an optimization program in MATLAB and you want to load your program's configuration data into a struct, `config`, with these fields:

```
% MATLAB
config.max_iter = 1000;
config.newmark.alpha = 0.25;
config.newmark.beta  = 0.5;
config.input_dir = "/xfer/sim_data/2022/05/28";
config.tune_coeff = [1.2e-4  -3.25  58.2];
```

One option is to store the above lines in a `.m` file then invoke the name of the file in your application. While convenient, this technique combines code and data—definitely not a best practice. Instead, let's store the data in YAML, `ini`, and TOML files then populate our `config` struct by loading these files with Python.

4.2 Reading YAML

Hierarchy in YAML is defined by horizonal whitespace, just as in Python. This YAML file

```
# optim_config.yaml
max_iter : 1000
newmark :
  alpha : 0.25
  beta : 0.5
input_dir : "/xfer/sim_data/2022/05/28"
tune_coeff : [1.2e-4, -3.25, 58.2]
```

loads into a dictionary like this in Python:

```
# Python
In : import yaml
In : with open('optim_config.yaml') as fh:
   :     config = yaml.safe_load(fh)
In : print(config)

{'max_iter': 1000, 'newmark': {'alpha': 0.25, 'beta': 0.5},
 'input_dir': '/xfer/sim_data/2022/05/28',
 'tune_coeff': [0.00012, -3.25, 58.2]}
```

Let's try it in MATLAB. If you haven't already, set up your MATLAB+Python environment (Chapter 3). Then try

```
% MATLAB
>> config = py.yaml.safe_load(py.open('optim_config.yaml'))
config =

  Python dict with no properties.

    {'max_iter': 1000, 'newmark': {'alpha': 0.25, 'beta': 0.5},
     'input_dir': '/xfer/sim_data/2022/05/28',
     'tune_coeff': [0.00012, -3.25, 58.2]}
```

It worked! Sort of—`config` is a Python dictionary within MATLAB, not a MATLAB struct. Individual values are accessible, but only in a clumsy manner:

```
% MATLAB
>> config.get('newmark').get('beta')

    0.5000
```

What we really want is a native MATLAB struct, not a Python dictionary. Enter `py2mat.m`. . . .

4.3 py2mat.m

`py2mat.m` is a generic Python-to-MATLAB variable converter I wrote to simplify MATLAB/Python interaction. We can use it to convert the Python dictionary returned by `yaml.safe_load()` to a native MATLAB struct:

```
% MATLAB
>> config = py2mat( py.yaml.safe_load(py.open('optim_config.yaml')) )

  struct with fields:

      max_iter: 1000
       newmark: [1x1 struct]
     input_dir: "/xfer/sim_data/2022/05/28"
    tune_coeff: {[1.2000e-04]  [-3.2500]  [58.2000]}
```

and now we can access fields the way we intended all along:

```
% MATLAB
>> config.newmark.beta

    0.5000
```

`py2mat.m` converts arbitrarily nested Python dictionaries, lists, tuples, sets, scalars, NumPy arrays, SciPy sparse matrices—even datetimes with correct timezone handling—to corresponding MATLAB structs and cell arrays containing scalars, strings, dense and sparse matrices, and MATLAB datetimes.

4.4 mat2py.m

The counterpart to `py2mat.m` is `mat2py.m`. It takes a native MATLAB variable and converts it to equivalent native Python variable within MATLAB. This is handy for passing MATLAB data into Python functions—as when using the Python `yaml` module to store a MATLAB variable in a YAML file. Let's give that a try.

4.5 df2t.m, t2df.m for Pandas Dataframes and MATLAB Tables

Do you work with Pandas dataframes or MATLAB tables? Artem Lenskiy wrote a pair of converters, `df2t.m` and `t2df.m`, that are the dataframe and table equivalents of `py2mat.m` and `mat2py.m`. You can find `df2t.m` and `t2df.m` at his PandasToMatlab Github project.

4.6 Writing YAML

The `dump()` function from the Python `yaml` module needs two things to write a YAML file: a variable whose contents are to be written and a file handle to write to. To do this from MATLAB however, we'll need to provide a *Python* variable and a *Python* file handle. We can get a Python version of the MATLAB variable with `mat2py()` and a Python file handle simply by calling `py.open()` instead of `open()`. This example echoes our current `config` variable to a new YAML file:

```
% MATLAB
>> fh = py.open('new_config.yaml','w');  % Python, not MATLAB, file handle
>> py_config = mat2py(config)

  Python dict with no properties.

      {'max_iter': 1000, 'newmark': {'alpha': 0.25, 'beta': 0.5},
       'input_dir': '/xfer/sim_data/2022/05/28',
       'tune_coeff': [0.00012, -3.25, 58.2]}

>> py.yaml.dump(py_config, fh)
>> fh.close()
```

The newly written file, `new_config.yaml`, looks like this:

```
input_dir: /xfer/sim_data/2022/05/28
max_iter: 1000
```

```
newmark:
  alpha: 0.25
  beta: 0.5
tune_coeff:
- 0.00012
- -3.25
- 58.2
```

While the sequence of entries and layout differ from the original `optim_config.yaml` file, the data loads into the same structure in both Python and MATLAB.

4.7 Reading TOML

TOML files can store hierarchical data in text files like YAML without need for correctly-aligned horizontal whitespace. Our configuration data can be stored in TOML like this:

```
# optim_config.toml
max_iter = 1000
input_dir = "/xfer/sim_data/2022/05/28"
tune_coeff = [1.2e-4, -3.25, 58.2]
[newmark]
alpha = 0.25
beta = 0.5
```

The `toml` module load function can take a file name directly so there's no need for a separate call to the file `open()`. The load function returns a dictionary:

```
# Python
In : import toml
In : config = toml.load('optim_config.toml')
In : print(config)
{'max_iter': 1000, 'input_dir': '/xfer/sim_data/2022/05/28',
 'tune_coeff': [0.00012, -3.25, 58.2],
 'newmark': {'alpha': 0.25, 'beta': 0.5}}
```

Loading TOML files in MATLAB is a one-line operation, just as it is in Python:

```
% MATLAB
>> config = py2mat(py.toml.load('optim_config.toml'))
```

```
  struct with fields:

      max_iter: 1000
     input_dir: "/xfer/sim_data/2022/05/28"
    tune_coeff: {[1.2000e-04]  [-3.2500]  [58.2000]}
       newmark: [1x1 struct]

>> config.newmark.alpha

    0.2500
```

4.8 Reading `ini`

`ini` files are less versatile than YAML or TOML since 1) they don't allow one to store arbitrarily nested hierarchical data and 2) all values come in as strings. An additional nuisance is that the read function in Python's `ini` handling module, `configparser`, does not throw an error if the given file cannot be read or parsed properly; the reader merely returns an empty list.

While somewhat resembling the TOML file, all values to the right of the equals sign are loaded as strings. Even `1.2e-4, -3.25, 58.2` is a single string that we have to deal with ourselves by separating the string into words then converting those words to numbers, Our configuration data might be stored in `ini` like so:

```
; optim_config.ini
[max_iter]
value = 1000
[newmark]
alpha = 0.25
beta = 0.5
[input_dir]
value = /xfer/sim_data/2022/05/28
[tune_coeff]
value = 1.2e-4, -3.25, 58.2
```

Loading this file in Python involves considerably more work than loading YAML or TOML.

```
# Python
In : import configparser
In : parser = configparser.ConfigParser()
In : parser.read('optim_config.ini')
In : max_iter = int(parser.get('max_iter', 'value'))
In : newmark_alpha = float(parser.get('newmark', 'alpha'))
In : newmark_beta  = float(parser.get('newmark', 'beta'))
In : input_dir  = parser.get('input_dir', 'value')
In : tune_coeff = [float(_) for _ in parser.get('tune_coeff', 'value').split(',')]

In : newmark_beta
Out: 0.5

In : tune_coeff
Out: [0.00012, -3.25, 58.2]
```

The MATLAB version is even messier:

```
% MATLAB
>> parser = py.configparser.ConfigParser();
>> parser.read('optim_config.ini');
>> max_iter = int64(py.int(parser.get('max_iter', 'value')));
>> newmark_alpha = double(py.float(parser.get('newmark', 'alpha')));
>> newmark_beta  = double(py.float(parser.get('newmark', 'beta' )));
>> input_dir  = string(parser.get('input_dir', 'value'));
>> tune_coeff = cellfun(@(x) double(x), ...
                        py2mat(parser.get('tune_coeff', 'value').split(',')));

>> newmark_beta

    0.5000

>> tune_coeff

    0.0001   -3.2500   58.2000
```

Chapter 5

Read and write SQLite databases in MATLAB with Python

2022/06/10

5.1 SQLite: the most widely deployed database on earth

Most likely the application you're using to read this article is running SQLite because SQLite is built into the Firefox, Chrome, and Safari web browsers. The computer you're on most definitely has it: SQLite is built into Android, iOS, macOS, Windows 10 and up, and is standard on Linux distributions. In addition to being everywhere, SQLite is also blazingly fast.

Given its ubiquity and high performance for storing and retrieving data, you may want to use SQLite in MATLAB. Your tool of choice for interacting with SQLite in MATLAB is MathWorks' Database Toolbox. If you have access to the Database Toolbox, the rest of this post may not interest you.

5.2 MATLAB + SQLite via Python

An alternative to the Database Toolbox is to use Python's `sqlite3` module in MATLAB.

In this article I'll explore the CRUD—create, read, update, delete—operations with SQLite in MATLAB with a simple database representing package deliveries between offices of a corporation. The database has a table that stores each package's type, weight, origin, and destination, and a second table with price per unit weight for each type. Here are represetative values:

`orders` table

origin	item	weight	destination
Z66	tube	1.73	N51
O64	package	3.66	J43
U74	tube	4.73	U71
K54	letter	1.14	V08
W09	letter	2.58	D50
T89	package	2.53	H15
P70	tube	2.58	E95
L96	card	4.20	N56

`rates` table

item	cost by weight
tube	0.46
package	0.65
letter	0.16
card	0.05

5.3 Create (and delete)

The "create" part of CRUD refers to three things: creating the database itself, creating tables within the database, and populating the tables with rows. An SQLite database is a file, typically saved with a `.db` extension, and is created the first time you attempt to "connect" to it. SQLite tables are created by sending SQL "create table" commands to the connected database, and popualted with SQL "insert" commands.

SQL table creation commands

Conventional SQL commands to create these two tables might look like this:

```
create table orders (origin text,
                     item   text,
                     weight float,
                     dest   text);
create table rates ( item   text,
                     cost_by_weight float);
```

If you put the six lines above in a text file called `make_deliveries_db.sql`, you can create a

SQLite database by piping the file into the `sqlite3` command line tool:

```
cat  make_deliveries_db.sql | sqlite3 deliveries.db      # Linux, macOS
type make_deliveries_db.sql | sqlite3.exe deliveries.db  # Windows
```

What we're really after though is doing this in MATLAB. To get there we'll start with a Python version since Python will be our gateway to using SQLite in MATLAB. A complete Python program to create the SQLite database file `deliveries.db` with our tables `orders` and `rates` is:

```python
#!/usr/bin/env python
import sqlite3
conn = sqlite3.connect('deliveries.db') # creates this if it does not exist
cursor = conn.cursor()
cursor.execute("drop table if exists orders;")
cursor.execute("drop table if exists rates;")
create_orders =   """create table orders (origin text,
                                          item   text,
                                          weight float,
                                          dest   text);"""
create_rates  =   """create table rates ( item   text,
                                          cost_by_weight float);"""
cursor.execute(create_orders)
cursor.execute(create_rates)
conn.commit()
conn.close()
```

The two "**`drop table if exists`**" lines delete the named tables from the database if they are already defined. If we didn't include these lines, the second time we try to run the program we'll hit a "table already exists" error. By dropping the tables (if they exist) first, we'll be able to rerun without errors—very handy during development.

Next: same thing in MATLAB. If you haven't already, set up your MATLAB+Python environment (Chapter 3). You'll note the MATLAB code looks almost the same has the Python equivalent above.

```matlab
% MATLAB
sqlite3 = py.importlib.import_module('sqlite3');
conn = sqlite3.connect('deliveries.db'); % creates this if it does not exist
cursor = conn.cursor();
cursor.execute("drop table if exists rates");
cursor.execute("drop table if exists orders");
create_order_table = "create table orders (" + ...
                                "origin text,"    + ...
                                "item   text,"    + ...
                                "weight float,"   + ...
                                "dest   text);";
cursor.execute(create_order_table);
cursor.execute("create table rates(item text, cost_by_weight float);");
conn.commit();
conn.close();
```

5.3.1 Inserts

Our tables are ready for content. Generic SQL commands to populate our two tables with the representative data shown above look like this:

```
insert into orders values('Z66', 'tube'   , 1.73, 'N51');
insert into orders values('O64',' package', 3.66, 'J43');
insert into orders values('U74',' tube'   , 4.73, 'U71');
insert into orders values('K54',' letter' , 1.14, 'V08');
insert into orders values('W09',' letter' , 2.58, 'D50');
insert into orders values('T89',' package', 2.53, 'H15');
insert into orders values('P70',' tube'   , 2.58, 'E95');
insert into orders values('L96',' card'   , 4.20, 'N56');

insert into rates values('tube'   , 0.46);
insert into rates values('package', 0.65);
insert into rates values('letter' , 0.16);
insert into rates values('card'   , 0.05);
```

Row inserts can be done with Python's `sqlite3` module either individually or in bulk. After you've created the `cursor` variable, the two forms of inserts can be done like this in Python:

```
# Python

# single row insert
row  =   [ 'Z66', 'tube'   , 1.73, 'N51']
cursor.execute('insert into orders values (?,?,?,?)', row)

# multiple row insert
rows = [ [ 'O64',' package', 3.66, 'J43'],
         [ 'U74',' tube'   , 4.73, 'U71'],
         [ 'K54',' letter' , 1.14, 'V08'],
         [ 'W09',' letter' , 2.58, 'D50'],
         [ 'T89',' package', 2.53, 'H15'],
         [ 'P70',' tube'   , 2.58, 'E95'],
         [ 'L96',' card'   , 4.20, 'N56'], ]
cursor.executemany('insert into orders values (?,?,?,?)', rows)
```

The multiple row insertion is done in a single transaction and yields much higher performance than single row inserts. The lines above look like this in MATLAB:

```
% MATLAB

% single row insert
row  =   { 'Z66', 'tube'   , 1.73, 'N51'};
cursor.execute('insert into orders values (?,?,?,?)', row);

% multiple row insert
rows = { { 'O64',' package', 3.66, 'J43'}, ...
         { 'U74',' tube'   , 4.73, 'U71'}, ...
         { 'K54',' letter' , 1.14, 'V08'}, ...
         { 'W09',' letter' , 2.58, 'D50'}, ...
         { 'T89',' package', 2.53, 'H15'}, ...
         { 'P70',' tube'   , 2.58, 'E95'}, ...
         { 'L96',' card'   , 4.20, 'N56'}, };
cursor.executemany('insert into orders values (?,?,?,?)', rows);
```

5.3.2 A data generator

A database whose biggest table has just eight rows underwhelms. This small MATLAB program creates a file, `orders.txt`, containing an arbitrary number of rows of random entries for the `orders` table (100,000 in this case). We can use it to produce a large database and test insert and query performance.

Aside: Python's Faker module is an excellent resource for generating arbitrary fake data for testing.

```
% MATLAB
% file: fake_data.m
n_rows = 100000;

item = {'letter', 'card', 'tube', 'package'};
n_item = length(item);
fh = fopen('orders.txt','w');
for i = 1:n_rows
```

```
    origin = char([64 + randi(26)  47 + randi(10,1,2)]);
    obj    = item{randi(n_item)};
    weight = 5*rand();
    dest   = char([64 + randi(26)  47 + randi(10,1,2)]);
    fprintf(fh,'%s %-8s  %5.2f %s\n', origin, obj, weight, dest);
end
fclose(fh);
```

Output is a text file, `orders.txt`, with four columns of random values. The first few lines of `orders.txt` might be

```
Y91 package    4.79 M81
R77 card       3.28 E70
H00 package    3.47 I90
L37 package    0.93 M46
S72 tube       3.28 E14
Y35 letter     3.76 G56
```

Order data saved to this text file needs to be loaded by our programs into a *row major* variable suitable for use by `cursor.executemany()`. In other words, the first level index i to the variable returns the i-th row of data. In Python this can be done with a list comprehension over lines of the file, where each line is separated into a list of values:

```
# Python
import pathlib
P = pathlib.Path('orders.txt')
rows = [_.split() for _ in P.read_text().split('\n') if _]
```

MATLAB's data loaders such as `textscan()` are column major so a bit of manipulation is needed to coerce the required structure:

```
% MATLAB
fh = fopen('orders.txt');
cols = textscan(fh, '%s %s %f %s');
fclose(fh);
n_rows = length(cols{1});
rows = cell(1,n_rows);
for r = 1:n_rows
    rows{r} = { cols{1}{r}, cols{2}{r}, cols{3}(r), cols{4}{r} };
end
```

Note the subtle indexing difference for column 3. Parentheses, rather than braces, must be used to dereference numeric values within a cell.

Complete programs to create and populate our deliveries database using data from `orders.txt` now look like this.

```
#!/usr/bin/env python
# file: make_db.py
import time
import sqlite3
import pathlib
conn = sqlite3.connect('deliveries.db') # creates if the file doesn't exist
cursor = conn.cursor()

# delete/create
cursor.execute("drop table if exists orders;")
cursor.execute("drop table if exists rates;")
cursor.execute("create table rates(item text, cost_by_weight float);");
create_orders_table = """create table orders (
                                origin text,
                                item   text,
                                weight float,
                                dest   text);"""
cursor.execute(create_orders_table)

rows = [ ['tube'    , 0.46],
         ['package', 0.65],
         ['letter' , 0.16],
         ['card'    , 0.05] ]
cursor.executemany('insert into rates values (?,?)', rows)

# load orders data from the text file
T_s = time.time()
P = pathlib.Path('orders.txt')
rows = [_.split() for _ in P.read_text().split('\n') if _]
print(f'Data read time = {time.time() - T_s} seconds')

T_s = time.time()
cursor.executemany('insert into orders values (?,?,?,?)', rows)
conn.commit()
print(f'Data insert time = {time.time() - T_s} seconds')

conn.close()
```

```
% MATLAB
% make_db.m
sqlite3 = py.importlib.import_module('sqlite3');
conn = sqlite3.connect('deliveries.db'); % create if it does not exist
cursor = conn.cursor();

% delete/create
cursor.execute("drop table if exists rates");
cursor.execute("drop table if exists orders");
cursor.execute("create table rates(item text, cost_by_weight float);");
create_order_table = "create table orders (" + ...
                          "origin text,"    + ...
                          "item   text,"    + ...
                          "weight float,"   + ...
                          "dest   text);";
cursor.execute(create_order_table);

% populate the rates table
rows = { {'tube'   , 0.46}, ...
         {'package', 0.65}, ...
         {'letter' , 0.16}, ...
         {'card'   , 0.05} };
cursor.executemany('insert into rates values (?,?)', rows);

% load orders data from the text file and insert to orders table
tic;
if (0)
    % BEFORE -- MATLAB file read + transpose
    fh = fopen('orders.txt');
    cols = textscan(fh, '%s %s %f %s');
    fclose(fh);
    n_rows = length(cols{1});
    rows = cell(1,n_rows);
    for r = 1:n_rows
        rows{r} = { cols{1}{r}, cols{2}{r}, cols{3}(r), cols{4}{r} };
    end
else
    % AFTER -- Python file read
    rows = py.load_orders.readfile('orders.txt');
end
fprintf('Data read time = %.6f seconds\n', toc)

tic;
cursor.executemany('insert into orders values (?,?,?,?)', rows);
conn.commit();
fprintf('Data insert time = %.6f seconds\n', toc)
```

```
conn.close();
```

5.3.3 Insert performance

Our programs' insert performance depends on how rapidly they can ingest the data from files and how long the bulk insert itself takes. Times in seconds on my Ubuntu 20.04 laptop using MATLAB 2022a and Python 3.8.8 are

n_rows	MATLAB read	MATLAB insert	Python read	Python insert
100,000	1.10	0.68	0.10	0.20
1,000,000	6.47	4.65	2.36	2.15

MATLAB is slower, but it must also do more work: it has to 1) transpose the loaded data from a column-major to a row-major container and 2) internally translate MATLAB strings and numeric values to Python equivalents when calling `cursor.executemany()`.

We can boost MATLAB's speed by using Python to ease both of these issues simply by replacing MATLAB's file reading code with the Python reader. The Python reader creates a Python-native variable that's immediately ready for the bulk insert with `cursor.executemany()`. This bypasses the MATLAB-to-Python impedance mismatch that MATLAB otherwise handles for us under the hood—at a time expense, of course.

Here's a small Python module with a function that implements the file loader we can call from MATLAB:

```python
# Python; file load_orders.py
# file load_orders.py
import pathlib
def readfile(File):
    P = pathlib.Path(File)
    return [_.split() for _ in P.read_text().split('\n') if _]
```

All that remains is to replace the MATLAB ingest code with a call to the above Python function. Here are the before and after MATLAB lines, where the 'before' code is blocked via `if (0)` and the 'after' code exists in the `else` block:

```matlab
% MATLAB
if (0)
    % BEFORE -- MATLAB file read + transpose
    fh = fopen('orders.txt');
    cols = textscan(fh, '%s %s %f %s');
    fclose(fh);
    n_rows = length(cols{1});
    rows = cell(1,n_rows);
    for r = 1:n_rows
        rows{r} = { cols{1}{r}, cols{2}{r}, cols{3}(r), cols{4}{r} };
    end
else
    % AFTER -- Python file read
    rows = py.load_orders.readfile('orders.txt');
end
```

MATLAB's performance with `load_orders.readfile()` is much closer to Python's:

n_rows	MATLAB+Python read	MATLAB insert	Python read	Python insert
100,000	1.10	0.68	0.10	0.20
1,000,000	3.11	2.34	2.36	2.15

5.4 Read

Database reads are done with the SQL `select` statement. Here's a query that computes the cummulative cost of shipments by originating office:

```
select origin,sum(weight)*cost_by_weight from orders O, rates R
  where O.item = R.item group by origin;
```

The query can be done programmatically in Python with

```
#!/usr/bin/env python
# file: select.py
import sqlite3
conn = sqlite3.connect('deliveries.db')
cursor = conn.cursor()

query = "select origin,sum(weight)*cost_by_weight from orders O, rates R" \
        "    where O.item = R.item group by origin order by origin;"
for row in cursor.execute(query).fetchall():
    print(f'{row[0]}  {row[1]:7.2f}')

conn.close()
```

and equivalently in MATLAB with

```
% MATLAB
% file: select.m
sqlite3 = py.importlib.import_module('sqlite3');
conn = sqlite3.connect('deliveries.db');
cursor = conn.cursor();

query = "select origin,sum(weight)*cost_by_weight from orders O, rates R" + ...
        "    where O.item = R.item group by origin order by origin;";
for row = cursor.execute(query).fetchall()
    fprintf('%s  %7.2f\n', string(row{1}{1}), row{1}{2});
end

conn.close()
```

Query performance is essentially the same in Python and MATLAB.

5.5 Update

A database update modifies the content of existing rows. Say the scale used to weigh packages (as opposed to the scale used for cards, letters, and tubes) was set to the wrong units, for example pounds instead of kilograms. This SQL update scales the weight of packages by 2.2 while leaving the wieghts of cards, letters, and tubes alone:

```
update orders set weight = weight*2.2 where item = 'package';
```

The programs below repeat the `select` statement from the previous section to show total cost by origin. Next the programs apply the weight update for packages then repeat the query. The programs below show how this is done programmatically in Python:

```
#!/usr/bin/env python
# file: select_update.py
import sqlite3
conn = sqlite3.connect('deliveries.db')
cursor = conn.cursor()

query = "select origin,sum(weight)*cost_by_weight from orders O, rates R" \
        "    where O.item = R.item group by origin order by origin;"
print('Original cost:')
for row in cursor.execute(query).fetchall():
    print(f'{row[0]}  {row[1]:7.2f}')
```

```
# update:  scale the weight of packages
cursor.execute("update orders set weight = weight*2.2 where item = 'package'")

print('After the weight correction:')
for row in cursor.execute(query).fetchall():
    print(f'{row[0]}  {row[1]:7.2f}')

conn.close()
```

and in MATLAB:

```
% MATLAB
% file: select_update.m
sqlite3 = py.importlib.import_module('sqlite3');
conn = sqlite3.connect('deliveries.db');
cursor = conn.cursor();

query = "select origin,sum(weight)*cost_by_weight from orders O, rates R" + ...
        "    where O.item = R.item group by origin order by origin;";
fprintf('Original cost:\n')
for row = cursor.execute(query).fetchall()
    fprintf('%s  %7.2f\n', string(row{1}{1}), row{1}{2});
end

% update:  scale the weight of packages
cursor.execute("update orders set weight = weight*2.2 where item = 'package'");

fprintf('After the weight correction:\n')
for row = cursor.execute(query).fetchall()
    fprintf('%s  %7.2f\n', string(row{1}{1}), row{1}{2});
end

conn.close()
```

Chapter 6

Create Excel files in MATLAB with Python's openpyxl module

2022/06/23

Your latest MATLAB masterpiece produced a goldmine of incisive data and now it's time to present the results to management. MATLAB Live Scripts are an excellent reporting mechanism but these may not suit an organization's workflow or documentation standards. Sometimes it's necessary to fall back to showing results in a basic tool like Microsoft Excel.

6.1 MATLAB + Excel = not easy

MATLAB's ability to create Excel files is modest at best. The `writecell()`, `writematrix()`, and `writetable()` functions can populate an Excel file with data, but only using the default text style. If you want to change typefaces, fonts, colors, merge cells and so on, you're in for some hardship. Custom formatting is only supported on Windows using the COM layer (see Write Data to Excel Spreadsheets, bottom of the page) and examples demonstrating this process are hard to find.

If you're using MATLAB on macOS or Linux, you're completely out of luck writing rich Excel files.

6.2 MATLAB + Python + Excel = easy!

If you include Python with your MATLAB setup (Chapter 3) on the otherhand, creating richly-formatted Excel files *on any platform* is easy. Excel files can be created in MATLAB by calling functions from the Python `openpyxl` module. The first step is to check if you have `openpyxl` and to install it if you don't.

6.3 Install openpyxl

Check your Python installation to see if it has openpyxl by starting the Python command line REPL and importing the module. If it imports successfully you can also check the version number:

```
> python
>>> import openpyxl
>>> openpyxl.__version__
'3.0.9'
```

If you get a ModuleNotFoundError, you'll need to install it. That can be done on an Anaconda Python installation with

```
> conda install openpyxl
```

or with

```
> python -m pip install openpyxl
```

using other Python distributions.

6.4 Start a new .xlsx file

The small program below creates an empty Excel file, my_analysis.xlsx, containing two worksheets, or tabs, called "Amazing Results" and "Summary". For special emphasis, the Summary tab's color is changed to green. Unfortunately colors in openpyxl must be provided as RGB or aRGB ('a' = alpha, to control transparency) hex values rather than words like "green".

Python file: empty.py:

```
#!/usr/bin/env python
import openpyxl as OP
book = OP.Workbook()
sheet_data = book.create_sheet("Amazing Results", 0)
sheet_summary = book.create_sheet("Summary", 1)
sheet_summary.sheet_properties.tabColor = "15AA15"
book.save("my_analysis.xlsx")
```

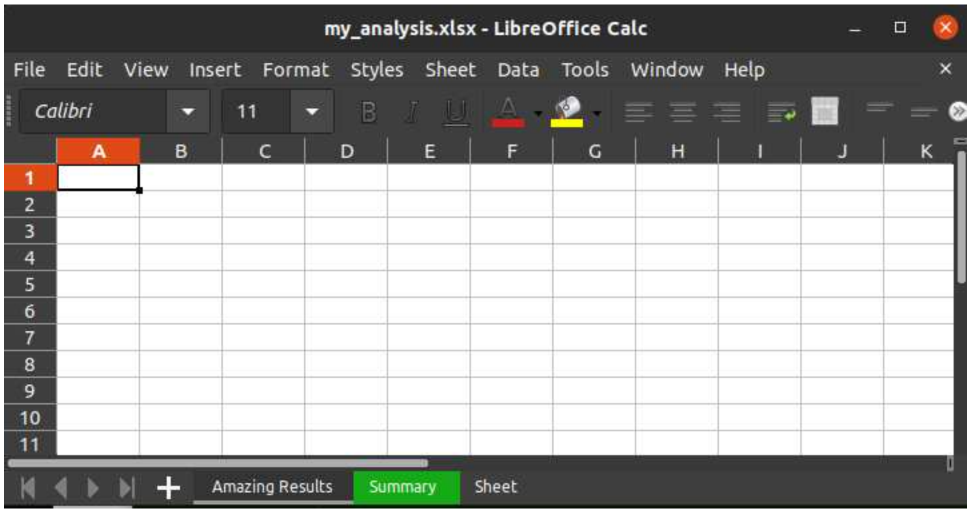

my_analysis.xlsx - LibreOffice Calc
File Edit View Insert Format Styles Sheet Data Tools Window Help
Calibri
11
Amazing Results
Summary
Sheet

Figure 6.1: `my_analysis.xlsx` created by `empty.py` or `empty.m`

The MATLAB version is `empty.m`:

```
% MATLAB
OP = py.importlib.import_module('openpyxl');
book = OP.Workbook();
sheet_data = book.create_sheet("Amazing Results", int64(0));
sheet_summary = book.create_sheet("Summary", int64(1));
sheet_summary.sheet_properties.tabColor = "15AA15";
book.save("my_analysis.xlsx");
```

The `create_sheet()` method's arguments are the worksheet's title and an optional position in the tab sequence, where 0 is the leftmost tab. Note the presence of the 'Sheet' tab. This was created by default and was pushed to the right when our sheets were added. The 'Sheet' tab can be deleted like this:

```
# Python
del book['Sheet']
   #  -  or  -
existing_sheet = book['Sheet']
book.remove_sheet(existing_sheet)
   #  -  or  -
existing_sheet = book.get_sheet_by_name('Sheet')
book.remove_sheet(existing_sheet)
```

MATLAB won't let us use `[ ]` to index the Python `book` object so only the third option works there:

```
% MATLAB
existing_sheet = book.get_sheet_by_name('Sheet');
book.remove_sheet(existing_sheet);
```

6.5 Avoiding bracket notation

As noted above, MATLAB won't let us index Python objects with brackets. This prevents us from using `openpyxl` shortcuts such as assigning a cell's value using something like `sheet['C8'] = 3.14159`. In Section 7.8 of *Python for MATLAB Development*, I suggested using a bridge module to overcome this interface limitation.

After the book came out, David Garrison at the MathWorks informed me that a bridge module is unnecessary since `openpyxl` offers method- and attribute-based ways to achieve the same thing. Setting cell `C8`'s value can also be done with

```
C8 = sheet.cell(3,8)
C8.value = 3.14159
```

All examples below use method and attribute access methods to accomodate MATLAB. The code examples therefore don't demonstrate the most idiomatic use of `openpyxl`, but they have the benefits that Python and MATLAB code samples look almost identical, and that the MATLAB implementation won't need a bridge module.

6.6 Fill a cell with a value

To populate a cell with a number or string, get a handle to the cell using the worksheet's `.cell()` method, pass it the 1-based row and column indices, then set the handle's `.value` attribute with the number or string of interest.

These lines put the string "Total" in cell B9 which corresponds to row 2, column 9:

```
# Python
B9 = sheet_data.cell(2, 9)
B9.value = "Total"
```

It looks like this in MATLAB:

```
% MATLAB
B9 = sheet_data.cell(int64(2), int64(9));
B9.value = "Total";
```

The `int64()` type casts are necessary because numeric literals in MATLAB have type `double` while the Python function expects integers.

6.7 Typeface size, font, color

A cell's font can be modified by creating a "font style" with the Font class then applying this style to the cell. The code example below creates a style, `ft_title`, having a 14 point blue Arial typeface with bold italics font.

```
# Python
import openpyxl.styles as styles
Font = styles.Font
ft_title = Font(name="Arial", size=14, color="0000FF",
                bold=True, italics=True)
```

MATLAB 2021b and earlier use `pyargs()` to set keyword value pairs:

```
% MATLAB 2021b and newer
ft_title = Font( pyargs("name","Arial", "size",int64(14),  ...
                        "color","0000FF", "bold",py.True,  ...
                        "italics",py.True) );
```

MATLAB 2022a and newer can use `pyargs()` as above or, even better, let you assign keyword values the same way as Python:

```
% MATLAB 2022a and newer
ft_title = Font(name=Arial, size=int64(14), color="0000FF", ...
                bold=py.True, italics=py.True);
```

The font style can then be applied to the text in a cell by setting the cell's `.font` attribute:

```
# Python
B9 = sheet_data.cell(2, 9)
B9.value = "Total"
B9.font  = ft_title
```

and

```
% MATLAB
B9 = sheet_data.cell(int64(2), int64(9));
B9.value = "Total";
B9.font  = ft_title;
```

6.8 Alignment

Text in a cell can be aligned vertically and horizontally, rotated, indented, and shrunk to fit a given container using the Alignment class. Text in merged cells looks good when it is centered both horizontally and vertically. That can be done with

```
# Python
import openpyxl.styles as styles
Alignment = styles.Alignment
B9.alignment = Alignment(horizontal="center", vertical="center")
```

and

```
% MATLAB 2021b and newer
B9.alignment = Alignment(pyargs("horizontal","center", ...
                                "vertical","center"));
```

```
% MATLAB 2022a and newer
B9.alignment = Alignment(horizontal="center", vertical="center");
```

6.9 Currency, percent, and other numeric formats

A cell can be formatted to show numeric values as currency or a percentage through the cell's `.number_format` attribute. The numerical format notation is rather unusual in that it allows positive values to be formatted diffently than negative values. If such a distinction is desired, a semicolon and the negative value format are appended to the postive value format. Financial ledgers sometimes show negative values in red and/or in parentheses so these variations are shown in the example below. Oddly, one may use words like 'Red' or 'Blue' to define the color within a `.number_format` but all other color definition options only allow hex values.

The `.number_format` attribute also controls date format styles and scientific notation. See the documentation for numeric styles for all options.

```
B9.number_format = '"$"#,##0.00;[Red]("$"#,##0.00)'  # eg  £12,345.67
B9.number_format = '0.0%;[Red]0.0%'                  # eg  98.7%
```

The dollar symbol can be replaced with other currency symbols simply by replacing `"$"` with the desired character — `"€"` for example.

6.10 Cell background color

Cell backgrounds can be set to a variety of colors and patterns with the PatternFill class. Following the same technique as with fonts and number styles, one first defines a pattern style then applies this to a cell's `.fill` attribute. The style `bg_green` defines a solid lime green background color:

```
# Python
import openpyxl.styles as styles
PatternFill = styles.PatternFill
bg_green = PatternFill(fgColor="C5FD2F", fill_type="solid")
B9 = sheet_data.cell(2,9)
B9.value = "Total"
B9.font  = ft_title
B9.fill  = bg_green
```

```
% MATLAB 2020b and newer
bg_green = PatternFill(pyargs("fgColor","C5FD2F","fill_type","solid"));
B9 = sheet_data.cell(int64(2),int64(9));
B9.value = "Total";
B9.font  = ft_title;
B9.fill  = bg_green;
```

```
% MATLAB 2022a and newer
bg_green = PatternFill(fgColor="C5FD2F", fill_type="solid");
B9 = sheet_data.cell(int64(2),int64(9));
B9.value = "Total";
B9.font  = ft_title;
B9.fill  = bg_green;
```

6.11 Merged cells

A block of adjacent cells can be coalessed into a single cell using a worksheet's `.merge_cells()` method. The method takes a cell block range as an argument. The example below merges the nine cells B2, B3, B4, C2, C3, C4, D2, D3, and D4 into one. After the merge, the cell's handle is accessed using the row and column coordinates of the top left cell:

```
# Python
sheet_data.merge_cells("B2:D4")
B2 = sheet_data.cell(2,2)
B2.value = "A Title"
B2.font = ft_title
B2.alignment = Alignment(horizontal="center", vertical="center")
```

```
% MATLAB 2020b and newer
sheet_data.merge_cells("B2:D4");
B2 = sheet_data.cell(int64(2),int64(2));
B2.value = "A Title";
B2.font = ft_title;
B2.alignment = Alignment(pyargs("horizontal","center", ...
                                "vertical","center");
```

```
% MATLAB 2022a and newer
sheet_data.merge_cells("B2:D4");
B2 = sheet_data.cell(int64(2),int64(2));
B2.value = "A Title";
B2.font = ft_title;
B2.alignment = Alignment(horizontal="center", vertical="center");
```

6.12 Equations

Equations are easily added to spreasheets as they are merely cells containing strings that begin with an equals sign and an operator. For example, to add an equation that sums values in the cell range E2:E50, one merely populates a cell with `'=SUM(E2:E50)'`.

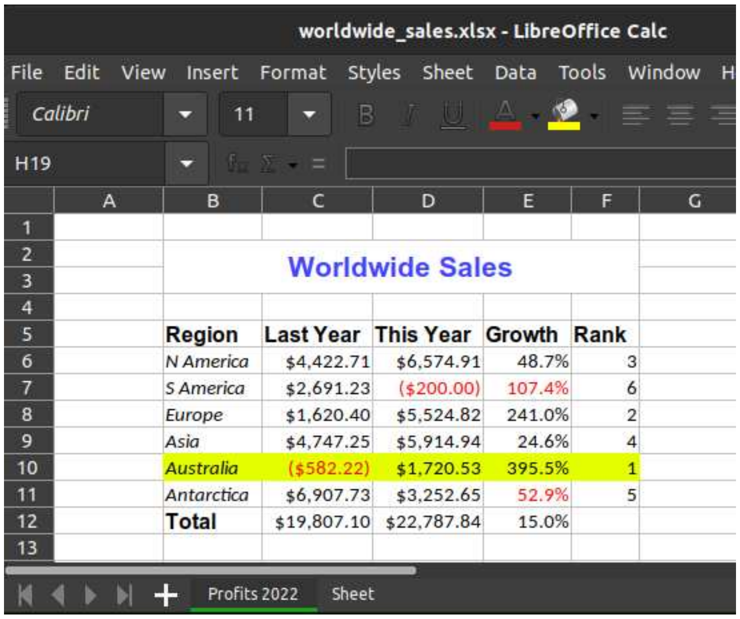
```
B9.value = '=SUM(E2:E50)'
```

6.13 Putting it all together

This spreadsheet includes all customizations described above:

File: all_together.py

```
#!/usr/bin/env python
import numpy as np
import openpyxl as OP
```


worldwide_sales.xlsx - LibreOffice Calc
File Edit View Insert Format Styles Sheet Data Tools Window H
Calibri
11
H19
Profits 2022
Sheet

Worldwide Sales

Region	Last Year	This Year	Growth	Rank
N America	$4,422.71	$6,574.91	48.7%	3
S America	$2,691.23	($200.00)	107.4%	6
Europe	$1,620.40	$5,524.82	241.0%	2
Asia	$4,747.25	$5,914.94	24.6%	4
Australia	($582.22)	$1,720.53	395.5%	1
Antarctica	$6,907.73	$3,252.65	52.9%	5
Total	$19,807.10	$22,787.84	15.0%	

Figure 6.2: `worldwide_sales.xlsx` created by `all_together.py` or `all_together.m`. Note: values are randomly generated and so will vary with each run.

```
import openpyxl.styles as styles
Font = styles.Font
Alignment = styles.Alignment
PatternFill = styles.PatternFill

book = OP.Workbook()
sheet_data = book.create_sheet("Profits 2022", 0)
sheet_data.sheet_properties.tabColor = "15AA15"
Region = ['N America', 'S America', 'Europe', 'Asia', 'Australia', 'Antarctica']
nR = len(Region)
profits = 10000 * (0.75 - np.random.rand(nR,2))
dollars = '"$"#,##0.00;[Red]("$"#,##0.00)'
pct     = '0.0%;[Red]0.0%'

sheet_data.merge_cells("B2:F3")
B2 = sheet_data.cell(2,2)
B2.value = 'Worldwide Sales'
B2.font = Font(name="Arial", size=16, color="4444FF", bold=True)
B2.alignment = Alignment(horizontal="center", vertical="center")

row, col = 5, 2

header_font = Font(name="Arial", size=12, bold=True)
```

```
region_cell = sheet_data.cell(row, 2)    # B5
region_cell.value = "Region"
region_cell.font  = header_font

last_year = sheet_data.cell(row, 3)      # C5
last_year.value = "Last Year"
last_year.font  = header_font

this_year = sheet_data.cell(row, 4)      # D5
this_year.value = "This Year"
this_year.font  = header_font

growth = sheet_data.cell(row, 5)         # E5
growth.value = "Growth"
growth.font  = header_font

rank = sheet_data.cell(row,6)            # F5
rank.value = 'Rank'
rank.font  = header_font

row += 1
highest_growth, highest_growth_row = -9.9e+99, 0
all_growth = np.zeros((nR,))
for r in range(nR):
    where      = sheet_data.cell(row+r, 2)
    where.value = Region[r]
    where.font = Font(name="Calibri", italic=True)
    last_year = sheet_data.cell(row+r, 3)
    this_year = sheet_data.cell(row+r, 4)
    growth    = sheet_data.cell(row+r, 5)
    last_year.value = profits[r,0]
    last_year.number_format = dollars
    this_year.value = profits[r,1]
    this_year.number_format = dollars
    growth.value = (this_year.value - last_year.value)/abs(last_year.value)
    all_growth[r] = growth.value
    growth.number_format = pct
    if growth.value > highest_growth:
        highest_growth = growth.value
        highest_growth_row = row + r

# highlight row with highest growth
highlight_color = PatternFill(fgColor="E4FF00",fill_type="solid")
for c in range(2,7):  # columns B to F
    cell = sheet_data.cell(highest_growth_row,c)
    cell.fill = highlight_color

# add values in the ranking column
rank = nR - all_growth.argsort().argsort()
for r in range(nR):
    cell = sheet_data.cell(row+r,6)
    cell.value = rank[r]

# cummulative row
total = sheet_data.cell(row+nR, 2)
total.value = "Total"
total.font  = header_font
for i,letter in enumerate(['C', 'D']):
    cell = sheet_data.cell(row+nR,i+3)
    cell.value = f'=SUM({letter}{row}:{letter}{row+nR-1})'
    cell.number_format = dollars

# cummulative growth
row += nR
cell = sheet_data.cell(row,5)
cell.value = f'=(D{row} - C{row})/ABS(C{row})'
cell.number_format = pct

book.save("worldwide_sales.xlsx")
```

MATLAB 2020b & newer using `pyargs()`:

File: `all_together_2020b.m`

```
% MATLAB 2020b & newer
clear
OP = py.importlib.import_module('openpyxl');
styles = py.importlib.import_module('openpyxl.styles');

Font = styles.Font;
Alignment = styles.Alignment;
PatternFill = styles.PatternFill;

book = OP.Workbook();
sheet_data = book.create_sheet("Profits 2022", int64(0));

sheet_data.sheet_properties.tabColor = "15AA15";
Region = {'N America', 'S America', 'Europe', 'Asia', 'Australia', 'Antarctica'};
nR = int64(length(Region));
profits = 10000 * (0.75 - rand(nR,2));
```

```
dollars = '"$"#,##0.00;[Red]("$"#,##0.00)';
pct     = '0.0%;[Red]0.0%';

sheet_data.merge_cells("B2:E3");
B2 = sheet_data.cell(int64(2),int64(2));
B2.value = 'Worldwide Sales';
B2.font = Font(pyargs("name","Arial","size",int64(16),...
                      "color","4444FF","bold",py.True));
B2.alignment = Alignment(pyargs("horizontal","center","vertical","center"));

row = int64(5);
col = int64(2);

header_font = Font(pyargs("name","Arial","size",int64(12),"bold",py.True));

region_cell = sheet_data.cell(row, int64(2));   % B5
region_cell.value = "Region";
region_cell.font  = header_font;

last_year = sheet_data.cell(row, int64(3));     % C5
last_year.value = "Last Year";
last_year.font  = header_font;

this_year = sheet_data.cell(row, int64(4));     % D5
this_year.value = "This Year";
this_year.font  = header_font;

growth       = sheet_data.cell(row, int64(5));  % E5
growth.value = "Growth";
growth.font  = header_font;

rank = sheet_data.cell(row, int64(6));          % F5
rank.value = "Rank";
rank.font  = header_font;

row = row + 1;
highest_growth = -9.9e+99;
highest_growth_row = int64(0);
all_growth = zeros(nR,1);
for r = 1:nR
    where = sheet_data.cell(row+r, int64(2));
    where.value = Region{r};
    where.font = Font(pyargs("name","Calibri","italic",py.True));
    last_year = sheet_data.cell(row+r, int64(3));
    this_year = sheet_data.cell(row+r, int64(4));
    growth    = sheet_data.cell(row+r, int64(5));
    last_year.value = profits(r,1);
    last_year.number_format = dollars;
    this_year.value = profits(r,2);
    this_year.number_format = dollars;
    growth.value = (this_year.value - last_year.value)/abs(last_year.value);
    all_growth(r) = growth.value;
    growth.number_format = pct;
    if growth.value > highest_growth
        highest_growth = growth.value;
        highest_growth_row = row + r;
    end
end

% highlight row with highest growth
highlight_color = PatternFill(pyargs("fgColor","E4FF00","fill_type","solid"));
for c = int64(2):int64(5)
    cell = sheet_data.cell(highest_growth_row,c);
    cell.fill = highlight_color;
end

% add values in the ranking column
[~, index] = sort(all_growth);
[~, rank] = sort(index);
rank = 1 + nR - int64(rank);
for r = 1:nR
    cell = sheet_data.cell(row+r,int64(6));
    cell.value = rank(r);
end

% cummulative row
row = row + int64(1);
total = sheet_data.cell(row+nR, int64(2));
total.value = "Total";
total.font  = header_font;
i = 0;
for letter = {'C', 'D'}
    cell = sheet_data.cell(row+nR,int64(i+3));
    cell.value = sprintf('=SUM(%s%d:%s%d)', ...
                        letter{1}, row, letter{1}, row+nR-1);
    cell.number_format = dollars;
    i = i + 1;
end

% cummulative growth
```

```
row = row + nR;
cell = sheet_data.cell(row,int64(5));
cell.value = sprintf('=(D%d - C%d)/ABS(C%d)', row, row, row);
cell.number_format = pct;

book.save("worldwide_sales.xlsx")
```

MATLAB 2022a & newer using `x=y` keyword argument assignments instead of `pyargs()`:

File: `all_together.m`

```
% MATLAB 2022a & newer
clear
OP = py.importlib.import_module('openpyxl');
styles = py.importlib.import_module('openpyxl.styles');

Font = styles.Font;
Alignment = styles.Alignment;
PatternFill = styles.PatternFill;

book = OP.Workbook();
sheet_data = book.create_sheet("Profits 2022", int64(0));

sheet_data.sheet_properties.tabColor = "15AA15";
Region = {'N America', 'S America', 'Europe', 'Asia', 'Australia', 'Antarctica'};
nR = int64(length(Region));
profits = 10000 * (0.75 - rand(nR,2));
dollars = '"$"#,##0.00;[Red]("$"#,##0.00)';
pct     = '0.0%;[Red]0.0%';

sheet_data.merge_cells("B2:E3");
B2 = sheet_data.cell(int64(2),int64(2));
B2.value = 'Worldwide Sales';
B2.font = Font(name="Arial", size=int64(16), color="4444FF", bold=py.True);
B2.alignment = Alignment(horizontal="center", vertical="center");

row = int64(5);
col = int64(2);

header_font = Font(name="Arial", size=int64(12), bold=py.True);

region_cell = sheet_data.cell(row, int64(2));   % B5
region_cell.value = "Region";
region_cell.font  = header_font;
```

```
last_year = sheet_data.cell(row, int64(3));     % C5
last_year.value = "Last Year";
last_year.font  = header_font;

this_year = sheet_data.cell(row, int64(4));     % D5
this_year.value = "This Year";
this_year.font  = header_font;

growth       = sheet_data.cell(row, int64(5));  % E5
growth.value = "Growth";
growth.font  = header_font;

rank = sheet_data.cell(row, int64(6));          % F5
rank.value = "Rank";
rank.font  = header_font;

row = row + 1;
highest_growth = -9.9e+99;
highest_growth_row = int64(0);
all_growth = zeros(nR,1);
for r = 1:nR
    where = sheet_data.cell(row+r, int64(2));
    where.value = Region{r};
    where.font = Font(name="Calibri", italic=py.True);
    last_year = sheet_data.cell(row+r, int64(3));
    this_year = sheet_data.cell(row+r, int64(4));
    growth    = sheet_data.cell(row+r, int64(5));
    last_year.value = profits(r,1);
    last_year.number_format = dollars;
    this_year.value = profits(r,2);
    this_year.number_format = dollars;
    growth.value = (this_year.value - last_year.value)/abs(last_year.value);
    all_growth(r) = growth.value;
    growth.number_format = pct;
    if growth.value > highest_growth
        highest_growth = growth.value;
        highest_growth_row = row + r;
    end
end

% highlight row with highest growth
highlight_color = PatternFill(fgColor="E4FF00",fill_type="solid");
for c = int64(2):int64(5)
    cell = sheet_data.cell(highest_growth_row,c);
    cell.fill = highlight_color;
```

```
end

% add values in the ranking column
[~, index] = sort(all_growth);
[~, rank] = sort(index);
rank = 1 + nR - int64(rank);
for r = 1:nR
    cell = sheet_data.cell(row+r,int64(6));
    cell.value = rank(r);
end

% cummulative row
row = row + int64(1);
total = sheet_data.cell(row+nR, int64(2));
total.value = "Total";
total.font  = header_font;
i = 0;
for letter = {'C', 'D'}
    cell = sheet_data.cell(row+nR,int64(i+3));
    cell.value = sprintf('=SUM(%s%d:%s%d)', ...
                         letter{1}, row, letter{1}, row+nR-1);
    cell.number_format = dollars;
    i = i + 1;
end

% cummulative growth
row = row + nR;
cell = sheet_data.cell(row,int64(5));
cell.value = sprintf('=(D%d - C%d)/ABS(C%d)', row, row, row);
cell.number_format = pct;

book.save("worldwide_sales.xlsx")
```

6.14 openpyxl can do more

Only a small fraction of openpyxl's capabilities are demonstrated here. This module can also create column filters, 2D and 3D stacked plots, 2D and 3D line plots with optional log scales, bar charts, scatter plots, pie charts, donut charts, radar plots, stock plots with high/low markers, 2D and 3D contour plots, and gauges.

Motivated MATLAB users will be able to make extraordinary Excel spreadsheets with openpyxl.

Chapter 7

Work with PostgreSQL in MATLAB Using the psycopg2 Python Module

2022/07/08

In Chapter 5 I showed how MATLAB can work with SQLite database files using the `sqlite3` module from the Python standard library. Here I'll demonstrate CRUD—create, read, update, delete—operations using the PostgreSQL database engine and the `psycopg2` Python module. The best way to work with PostgreSQL in MATLAB is to use the MathWorks' Database Toolbox. This post may only interest you if you lack access to the Database Toolbox and you're willing to include Python in your MATLAB workflow (Chapter 3).

7.1 Install psycopg2

Unlike `sqlite3`, `psycopg2` is not in the standard Python library. Check your Python installation to see if it has `psycopg2` by starting the Python command line REPL and importing the module. If it imports successfully you can also check the version number:

```
> python
>>> import psycopg2
>>> psycopg2.__version__
'2.9.3 (dt dec pq3 ext lo64)'
```

If you get a `ModuleNotFoundError`, you'll need to install it. That can be done on an Anaconda Python installation with

```
> conda install psycopg2
```

or with

```
> python -m pip install psycopg2
```

using other Python distributions.

7.2 Connecting to a PostgreSQL Database

The most challenging step in this post is setting up, configuring, and making an initial connection to a PostgreSQL database service—all of which are beyond the scope of this post. Ideally you are already familiar with the process, or have access to an existing database service, or know someone who can get you set up. If not, follow the PostgreSQL getting started documentation. The rest of this post assumes you have

- an account and password
- permission to create and delete databases
- network access

on or to a PostgreSQL server. Briefly, a PostgreSQL server administrator on a Linux computer might issue commands such as

```
> sudo -u postgres psql
postgres=# create role Al with login password 'SuperSecret';
postgres=# create database deliveries;
postgres=# grant all privileges on database deliveries to Al;
postgres=# quit
```

to set me up.

Run this small Python program to check if you can connect to your PostgreSQL server:

```
#!/usr/bin/env python
# test_pg_connection.py
import psycopg2
conn = psycopg2.connect(host="big_db_server", port=5432, # default port
                        user="Al", password="SuperSecret",
                        database="deliveries")
conn.close()
```

Of course change the hostname, username, password, and/or port number as appropriate for your setup. All is well if the program runs without errors. If there are errors, they'll likely be one of these:

- `psycopg2.OperationalError: could not connect to server: Connection refused`
- `psycopg2.OperationalError: FATAL: password authentication failed for user Al`
- `psycopg2.OperationalError: FATAL: database deliveries does not exist`

The first means either the database service is down, or a network issue (such as a firewall) is preventing access. The second and third errors show the database service is reachable, but the account or database you want to use either does not exist or is not configured properly. You or your database administrator will need to reexamine the setup.

7.3 Create (and delete)

As hinted by the `deliveries` database in the `SQL` commands above, I'll repeat the package delivery example from the earlier MATLAB+SQLite chapter. To start, I'll run the same `fake_data.m` program in MATLAB (source listing in section 5.3.2) to create a text file, `orders.txt`, containing 100,000 lines of data to insert into the database.

MATLAB:

```
>> fake_data
>>
```

A Python program that reads `orders.txt`, creates and inserts the values into the orders table in batches of 10,000 rows at a time looks like this:

```python
#!/usr/bin/env python
# file: make_pg_db.py
import time
import psycopg2
import pathlib
conn = psycopg2.connect(host="localhost", port=5432,
                        user="Al", password="SuperSecret",
                        database="deliveries")

cursor = conn.cursor()
```

```
# delete/create
cursor.execute("drop table if exists orders;")
cursor.execute("drop table if exists rates;")
cursor.execute("""create table rates(item varchar primary key,
                                     cost_by_weight float);""")
create_orders_table = """create table orders (
                              origin varchar,
                              item   varchar,
                              weight float,
                              dest   varchar);"""
cursor.execute(create_orders_table)

rows = [ ['tube'   , 0.46],
         ['package', 0.65],
         ['letter' , 0.16],
         ['card'   , 0.05] ]
insert = "insert into rates (item,cost_by_weight) values(%s,%s)"
cursor.executemany(insert, rows)

# load orders data from the text file
T_s = time.time()
P = pathlib.Path('orders.txt')
rows = [_.split() for _ in P.read_text().split('\n') if _]
print(f'Data read time = {time.time() - T_s:.3f} seconds')

insert = "insert into orders (origin,item,weight,dest) values(%s,%s,%s,%s)"
T_s = time.time()
batch_size = 10_000 # insert this many rows at a time
for i in range(0, len(rows), batch_size):
    rows_subset = rows[i:i+batch_size]
    cursor.executemany(insert, rows_subset)
    print(f'inserted {i+1} to {i+batch_size}')
conn.commit()
print(f'Data insert time = {time.time() - T_s:.3f} seconds')

conn.close()
```

I'm connecting to a PostgreSQL instance running on the same computer (2015 Dell XPS 13 laptop running Ubuntu 20.04) where I run `make_pg_db.py`. Insert performance is about 8,000 rows per second, 50x slower that SQLite inserts. I'm sure there are PostgreSQL configuration optimizations I could make to increase insert performance if I knew more about this database engine.

As with inserts to a SQLite database, MATLAB needs to store its insertion data row-major. We already saw that reading and reshuffling `orders.txt` from column-major to row-major in MATLAB is slow (section 5.3.2) so our MATLAB implementation of the PostgreSQL insert program will use the same `load_orders.py` Python module used to speed up MATLAB in the SQLite example.

Two equivalent MATLAB insertion program appear below. The first relies on the Python function keyword argument enhancements in 2022a.

```
% MATLAB 2022a and newer
% file: make_pg_db.m
% make_pg_db.m
psycopg2 = py.importlib.import_module('psycopg2');
conn = psycopg2.connect(host="localhost", port=int64(5432), ...
                        user="Al", password="SuperSecret",  ...
                        database="deliveries");
cursor = conn.cursor();

% delete/create
cursor.execute("drop table if exists rates");
cursor.execute("drop table if exists orders");
cursor.execute("create table rates(item varchar primary key, " + ...
                                   "cost_by_weight float);");
create_orders_table = "create table orders (" + ...
                           "origin varchar,"  + ...
                           "item   varchar,"  + ...
                           "weight float,"    + ...
                           "dest   varchar);";
cursor.execute(create_orders_table);

% populate the rates table
rows = { {'tube'    , 0.46}, ...
         {'package', 0.65}, ...
         {'letter' , 0.16}, ...
         {'card'    , 0.05} };
insert = "insert into rates (item,cost_by_weight) values(%s,%s)";
cursor.executemany(insert, rows);

% load orders data from the text file and insert to orders table
tic;
rows = py.load_orders.readfile('orders.txt');
fprintf('Data read time = %.6f seconds\n', toc)

insert = "insert into orders (origin,item,weight,dest) values(%s,%s,%s,%s)";
tic
batch_size = 10000; % insert this many rows at a time
n_rows = length(rows);
for i = 1: batch_size:n_rows
```

```
    i_end = min(i+batch_size-1, n_rows);
    rows_subset = rows(i:i_end);
    cursor.executemany(insert, rows_subset)
    fprintf('inserted %d to %d\n', i, i_end);
end
conn.commit();
fprintf('Data insert time = %.6f seconds\n', toc)
conn.close();
```

while the version below uses the older `pyargs()` function to wrap Python keyword arguments. It works with MATLAB 2020b and newer. The two versions of this program differ only at lines 4, 5 and 6.

```
% MATLAB 2020b and newer
% file: make_pg_db_2020b.m
psycopg2 = py.importlib.import_module('psycopg2');
conn = psycopg2.connect(pyargs("host","localhost","port",int64(5432), ...
                               "user","Al","password","SuperSecret",  ...
                               "database","deliveries"));
cursor = conn.cursor();

% delete/create
cursor.execute("drop table if exists rates");
cursor.execute("drop table if exists orders");
cursor.execute("create table rates(item varchar primary key, " + ...
                                   "cost_by_weight float);");
create_orders_table = "create table orders (" + ...
                           "origin varchar,"  + ...
                           "item   varchar,"  + ...
                           "weight float,"    + ...
                           "dest   varchar);";
cursor.execute(create_orders_table);

% populate the rates table
rows = { {'tube'    , 0.46}, ...
         {'package', 0.65}, ...
         {'letter' , 0.16}, ...
         {'card'    , 0.05} };
insert = "insert into rates (item,cost_by_weight) values(%s,%s)";
cursor.executemany(insert, rows);

% load orders data from the text file and insert to orders table
tic;
rows = py.load_orders.readfile('orders.txt');
fprintf('Data read time = %.6f seconds\n', toc)
```

```
insert = "insert into orders (origin,item,weight,dest) values(%s,%s,%s,%s)";
tic
batch_size = 10000; % insert this many rows at a time
n_rows = length(rows);
for i = 1: batch_size:n_rows
    i_end = min(i+batch_size-1, n_rows);
    rows_subset = rows(i:i_end);
    cursor.executemany(insert, rows_subset)
    fprintf('inserted %d to %d\n', i, i_end);
end
conn.commit();
fprintf('Data insert time = %.6f seconds\n', toc)
conn.close();
```

MATLAB insert performance matches Python's, about 8,000 rows per second—in other words, not great compared to SQLite inserts.

7.4 Read

This query computes the cummulative cost of shipments by originating office.

```
select origin,sum(weight*cost_by_weight) from orders O, rates R
  where O.item = R.item group by origin;
```

Programmatic implementations of the query in Python and MATLAB resemble those in the SQLite select example:

```
#!/usr/bin/env python
# file: select_pg.py
import psycopg2
conn = psycopg2.connect(host="localhost", port=5432,
                        user="Al", password="SuperSecret",
                        database="deliveries")
cursor = conn.cursor()

query = "select origin,sum(weight*cost_by_weight) from orders O, rates R" \
        "    where O.item = R.item group by origin order by origin;"
cursor.execute(query)
for row in cursor.fetchall():
    print(f'{row[0]}  {row[1]:7.2f}')
```

```
conn.close()
```

and equivalently in MATLAB 2022a+ with

```
% MATLAB 2022a and newer
% file: select_pg.m
psycopg2 = py.importlib.import_module('psycopg2');
conn = psycopg2.connect(host="localhost", port=int64(5432), ...
                        user="Al", password="SuperSecret",  ...
                        database="deliveries");
cursor = conn.cursor();

query = "select origin,sum(weight*cost_by_weight) from orders O, rates R" + ...
        "    where O.item = R.item group by origin order by origin;";
cursor.execute(query);
for row = cursor.fetchall()
    fprintf('%s  %7.2f\n', string(row{1}{1}), row{1}{2});
end

conn.close()
```

and in MATLAB 2020b+ with

```
% MATLAB 2020b and newer
% select_pg.m
psycopg2 = py.importlib.import_module('psycopg2');
conn = psycopg2.connect(pyargs("host","localhost","port",int64(5432), ...
                               "user","Al","password","SuperSecret",  ...
                               "database","deliveries"));
cursor = conn.cursor();

query = "select origin,sum(weight*cost_by_weight) from orders O, rates R" + ...
        "    where O.item = R.item group by origin order by origin;";
cursor.execute(query);
for row = cursor.fetchall()
    fprintf('%s  %7.2f\n', string(row{1}{1}), row{1}{2});
end

conn.close()
```

7.5 Update

As in the SQLite update example, this SQL update scales the weight of packages by 2.2 while leaving the wieghts of cards, letters, and tubes alone:

```
update orders set weight = weight*2.2 where item = 'package';
```

The programs below repeat the `select` statement from the previous section to show total cost by origin. The SQL `update` is implemente with line 17 of the Python program and lines 18 in the two MATLAB versions. After updating the `weight` field the programs re-query for the new cummulative costs:

```
#!/usr/bin/env python
# file: select_update_pg.py
import psycopg2
conn = psycopg2.connect(host="localhost", port=5432,
                        user="Al", password="SuperSecret",
                        database="deliveries")
cursor = conn.cursor()

query = "select origin,sum(weight*cost_by_weight) from orders O, rates R" \
        "    where O.item = R.item group by origin order by origin;"
print('Original cost:')
cursor.execute(query)
for row in cursor.fetchall():
    print(f'{row[0]}  {row[1]:7.2f}')

# update:  scale the weight of packages
cursor.execute("update orders set weight = weight*2.2 where item = 'package'")

print('After the weight correction:')
cursor.execute(query)
for row in cursor.fetchall():
    print(f'{row[0]}  {row[1]:7.2f}')

conn.close()
```

MATLAB 2022a+:

```
% MATLAB 2022a and newer
% file: select_update_pg.m
psycopg2 = py.importlib.import_module('psycopg2');
```

```
conn = psycopg2.connect(pyargs("host","localhost","port",int64(5432), ...
                               "user","Al","password","SuperSecret",  ...
                               "database","deliveries"));
cursor = conn.cursor();

query = "select origin,sum(weight*cost_by_weight) from orders O, rates R" + ...
        "    where O.item = R.item group by origin order by origin;";
fprintf('Original cost:\n')
cursor.execute(query);
for row = cursor.fetchall()
    fprintf('%s  %7.2f\n', string(row{1}{1}), row{1}{2});
end

% update:  scale the weight of packages
cursor.execute("update orders set weight = weight*2.2 where item = 'package'");

fprintf('After the weight correction:\n')
cursor.execute(query);
for row = cursor.fetchall()
    fprintf('%s  %7.2f\n', string(row{1}{1}), row{1}{2});
end

conn.close()
```

MATLAB 2020b+:

```
% MATLAB 2020b and newer
% file: select_update_pg_2020b.m
psycopg2 = py.importlib.import_module('psycopg2');
conn = psycopg2.connect(pyargs("host","localhost","port",int64(5432), ...
                               "user","Al","password","SuperSecret",  ...
                               "database","deliveries"));
cursor = conn.cursor();

query = "select origin,sum(weight*cost_by_weight) from orders O, rates R" + ...
        "    where O.item = R.item group by origin order by origin;";
fprintf('Original cost:\n')
cursor.execute(query);
for row = cursor.fetchall()
    fprintf('%s  %7.2f\n', string(row{1}{1}), row{1}{2});
end

% update:  scale the weight of packages
cursor.execute("update orders set weight = weight*2.2 where item = 'package'");
```

```
fprintf('After the weight correction:\n')
cursor.execute(query);
for row = cursor.fetchall()
    fprintf('%s  %7.2f\n', string(row{1}{1}), row{1}{2});
end

conn.close()
```

Chapter 8

Work with MongoDB in MATLAB Using the pymongo Python Module

2022/07/22

Earlier I showed how Python can help MATLAB interact with SQLite (Chapter 5) and PostgreSQL (Chapter 7). I'll conclude the database portion of this series with examples showing how to work with MongoDB in MATLAB using the Python module `pymongo`. As with the other databases, the best way to interact with MongoDB in MATLAB is to use the MathWorks' Database Toolbox. This post may only be useful to you if you can include Python in your MATLAB workflow (Chapter 3) and don't have access to the Database Toolbox.

8.1 Install `pymongo`

You'll need the `pymongo` Python module in your installation to call it from MATLAB. Check your Python installation to see if it has `pymongo` by starting the Python command line REPL and importing the module. If it imports successfully you can also check the version number:

```
> python
>>> import pymongo
>>> pymongo.__version__
'3.11.0'
```

If you get a `ModuleNotFoundError`, you'll need to install it. That can be done on an Anaconda Python installation with

```
> conda install pymongo
```

or with

```
> python -m pip install pymongo
```

using other Python distributions.

8.2 Connecting to a MongoDB Database

As with the PostgreSQL post, probably the hardest step in this post involves making the initial connection to a MongoDB server. That will only be successful if you have

- an account and password
- permission to create and delete databases
- network access to the MongoDB server (if it isn't on your local computer) on or to a MongoDB server.

To keep things simple, here I'll work with a MongoDB instance running on my localhost. Refer to instructions on starting a MongoDB database on Ubuntu, Red Hat, Windows, macOS if you're unfamiliar with the process. The `mongo` command line tool shows that the database engine is running and that I'm able to connect to it:

```
> mongo
MongoDB shell version v3.6.8
connecting to: mongodb://127.0.0.1:27017
Implicit session: session { "id" : UUID("e629be9e-e016-4e37-872c-e3f4ea3eb739") }
MongoDB server version: 3.6.8
Server has startup warnings:
2022-07-08T09:40:01.686-0700 I STORAGE  [initandlisten]
2022-07-08T09:40:03.337-0700 I CONTROL  [initandlisten]
2022-07-08T09:40:03.337-0700 I CONTROL  [initandlisten] ** WARNING: Access control is not enabled for the database.
2022-07-08T09:40:03.337-0700 I CONTROL  [initandlisten] **          Read and write access to data and configuration
2022-07-08T09:40:03.337-0700 I CONTROL  [initandlisten]

> show dbs;
admin           0.000GB
config          0.000GB
local           0.000GB
```

Connecting manually works so I'll try it programmatically, first in Python, then in MATLAB:

Python:

```
#!/usr/bin/env python
# file: mongo_connect.py
from pymongo import MongoClient
client = MongoClient('localhost:27017')
for db in client.list_databases():
    # db is a dictionary with entries like
    #   {'name': 'admin', 'sizeOnDisk': 32768.0, 'empty': False}
    print(db)
    print(db['name'])
client.close()
```

Running this gives the same result as `show dbs;` in the mongo shell:

```
> ./mongo_connect.py
admin
config
local
```

Now in MATLAB:

```
% file: mongo_connect.m
pymongo = py.importlib.import_module('pymongo');
client = pymongo.MongoClient('localhost:27017');
for db = py.list(client.list_databases())
    fprintf('%s\n', string(db{1}{'name'}))
end
client.close()
```

The MATLAB version is a bit more complex for a couple reasons:

1. The `for` loop cannot iterate over `client.list_databases()` directly as Python can. Attempting to do that results in the error `Array formation and parentheses-style indexing with objects of class 'py.pymongo.command_cursor.CommandCursor' is not allowed. Use objects of class 'py.pymongo.command_cursor.CommandCursor' only as scalars or use a cell array. Error in mongo_connect` The work-around is to explicitly expand this function call to a Python list.
2. The loop variable `db` is a single-item Python list containing the dictionary we expect rather than the dictionary itself. The {1} index before the {`'name'`} key retrieves the dictionary out of the list. Next, the database name returned by `db{1}{'name'}` is a a Python string, but if we attempt to `fprintf()` that directly, MATLAB will print the individual characters of the string. To get around that we have to explicitly convert the Python string to a MATLAB version so we end up with `string(db{1}{'name'})`.

MATLAB 2020b and higher then give the expected output:

```
>> mongo_connect
admin
config
local
```

While relatively simple, the issues we're stumbling across using MongoDB in MATLAB with `pymongo` foreshadow more challenges ahead.

8.3 Aside: connecting to a remote MongoDB Database

The client connection to a local MongoDB instance done with

```
client = MongoClient('localhost:27017')
```

takes a notably different form when connecting to a remote database that requires authentication. In that case the connection string passed to the `MongoClient()` function is more involved:

Python:

```
db_user = 'al'
db_passwd = 'SuperSecret'
db_host = '413.198.231.158'
db_port = 27027
uri = "mongodb://%s:%s@%d/purchase_orders" % (
         db_user, db_passwd, db_host, db_port)
client = MongoClient(uri)
```

MATLAB 2020b and higher:

```
db_user = 'al';
db_passwd = 'SuperSecret';
db_host = '413.198.231.158';
db_port = 27027;
uri = sprintf("mongodb://%s:%s@%s:%d/purchase_orders", ...
        db_user, db_passwd, db_host, db_port);
client = pymongo.MongoClient(uri);
```

8.4 Insert

I'll insert the same package delivery data used for the SQLite+MATLAB (Chapter 5) and PostgreSQL+MATLAB (Chapter 7) examples. Well, not exactly the same since NoSQL database engines such as MongoDB don't have a concept of tables. Rather than separating the delivery data into `orders` and `rates` tables, all information will be inserted into *documents* (BSON records) of a *collection*.

The first rows of each SQL table in the SQLite example are

`orders` table

origin	item	weight	destination
Z66	tube	1.73	N51

`rates` table

item	cost by weight
tube	0.46

These can be represented by the following JSON record (JSON is a subset of BSON):

```
{ "origin" : "Z66",
  "item" : "tube",
  "weight" : 1.73,
  "dest" : "N51",
  "cost_by_weight" : 0.46}
```

8.4.1 Inserting a single record

Creating a collection and inserting a document takes just a few lines of code. This is an expanded form of the connection tester which inserts the record above:

Python:

```python
#!/usr/bin/env python
# file: mongo_insert_one.py
from pymongo import MongoClient
client = MongoClient('localhost:27017')
```

```
db = client.deliveries
record =  { "origin" : "Z66", "item" : "tube",
            "weight" : 1.73, "dest" : "N51",
            "cost_by_weight" : 0.46}
db.deliveries.insert_one( record )
client.close()
```

Note that I didn't pass a JSON string to the `.insert_one()` method. Instead, I gave it a Python dictionary. This is a convenience provided by the `pymongo` module which converts dictionaries to JSON—or BSON if needed—for us. Once again the MATLAB version requires more effort as it won't create the `deliveries` collection for us automatically as in Python. Here we have to explicitly spell out a database and a collection to create:

MATLAB:

```
% file: mongo_insert_one.m
pymongo = py.importlib.import_module('pymongo');
client = pymongo.MongoClient('localhost:27017');
deliveries = client.get_database('deliveries').get_collection('deliveries');
record = py.dict(pyargs( ...
                 "origin", "Z66", "item", "tube", ...
                 "weight", 1.73, "dest", "N51",   ...
                 "cost_by_weight", 0.46));
deliveries.insert_one( record )
client.close()
```

Now that we've covered the basic mechanics of connecting to a MongoDB database and inserting a record we can move on to a more realistic program that reads many records from a text file then inserts them in bulk.

8.4.2 Inserting multiple records

Once again I'll reuse the `fake_data.m` m-file from the SQLite chapter (source listing in section 5.3.2) to create a text file of delivery data. Running it in MATLAB,

```
>> fake_data
>>
```

leaves me with the text file, `orders.txt`, which has 100,000 lines of delivery orders.

I'd also like to reuse the four line Python module `load_orders.py` to speed up data ingest in MATLAB as I did for the SQLite and PostgreSQL examples but the output from its `readfile()` function isn't well-suited for use with MongoDB. We saw that `pymongo`'s `insert_one()` method takes a Python dictionary as an input argument. The corresponding bulk insert method, `insert_many()`, takes a list of dictionaries. The `load_orders.readfile()` function returns a list of lists which would have to be manipulated further to a list of dictionaries. Rather than doing that, I'll implement a new file reader that directly returns the structure I want. It looks like this:

Python:

```
# file: load_orders_for_mongodb.py
import pathlib
def readfile(File):
    cbw = {'tube'   : 0.46, 'package': 0.65,
           'letter' : 0.16, 'card'   : 0.05}
    P = pathlib.Path(File)
    all_orders = []
    rows = [_.split() for _ in P.read_text().split('\n') if _]
    for origin, item, weight, dest in rows:
        order = { 'origin' : origin, 'item' : item,
                  'weight' : float(weight), 'dest' : dest,
                  'cost_by_weight' : cbw[item] }
        all_orders.append( order )
    return all_orders
```

I'll import this module and call its `readfile()` function in both Python and MATLAB bulk insert programs:

Python:

```
#!/usr/bin/env python
# file: mongo_insert_many.py
from pymongo import MongoClient
from load_orders_for_mongodb import readfile
import time
client = MongoClient('localhost:27017')
db = client.deliveries

tic = time.time()
all_records = readfile('orders.txt')
toc = time.time() - tic
print(f'read   {len(all_records)} took {toc:.3f} s')

# delete the deliveries collection if it already exists
if 'deliveries' in db.list_collection_names():
    db['deliveries'].drop()
```

```
tic = time.time()
db.deliveries.insert_many( all_records )
toc = time.time() - tic
print(f'insert {len(all_records)} took {toc:.3f} s')

client.close()
```

MATLAB:

```
% file: mongo_insert_many.m
pymongo = py.importlib.import_module('pymongo');
LO = py.importlib.import_module('load_orders_for_mongodb');
client = pymongo.MongoClient('localhost:27017');
db = client.get_database('deliveries');
deliveries = db.get_collection('deliveries');

tic;
all_records = LO.readfile('orders.txt');
fprintf('read   %d took %.3f s\n', length(all_records), toc)

% delete the deliveries collection if it already exists
if db.list_collection_names().count('deliveries') > 0
    db.drop_collection('deliveries');
end

tic;
deliveries.insert_many( all_records );
fprintf('insert %d took %.3f s\n', length(all_records), toc)

client.close()
```

Insert performance is fairly impressive. (Times in seconds; hardware is a 2015 Dell XPS 13 laptop running Ubuntu 20.04, Anaconda Python 2021.05, and MATLAB 2020b)

n_rows	MATLAB + Python read	MATLAB insert	Python read	Python insert
100,000	0.33	1.41	0.18	1.30
1,000,000	2.89	15.15	2.23	14.87

8.5 Query

The SQL statement

```
select origin,sum(weight*cost_by_weight) from orders O, rates R
  where O.item = R.item group by origin;
```

used in the SQLite and PostgreSQL examples takes a starkly different form in MongoDB. There the query can be expressed as

```
db.deliveries.aggregate([{$group :
    {_id : {F1:'$origin', F2:'$dest'},
     tot : {$sum : '$weight'} } } ])
```

which can be implemented like this in Python:

Python:

```
#!/usr/bin/env python
# file: mongo_query.py
from pymongo import MongoClient
client = MongoClient('localhost:27017')
db = client.deliveries

n_records = db.deliveries.count_documents({})
print(f'number of records in deliveries = {n_records}')

pipeline = [{'$group' : {'_id' : {'F1':'$origin', 'F2':'$dest'},
                         'tot' : {'$sum' : '$weight'}}}]

for x in db.deliveries.aggregate(pipeline=pipeline):
    origin = x['_id']['F1']
    dest   = x['_id']['F2']
    sum_weight = x['tot']
    print(f'{origin} -> {dest}   {sum_weight:6.2f}')

client.close()
```

Lines 10-13 in the MATLAB code below show the rather clumsy notation needed to express the multiply-nested dictionaries of the aggregation pipeline:

MATLAB:

```
% file: mongo_query.m
pymongo = py.importlib.import_module('pymongo');
client = pymongo.MongoClient('localhost:27017');
deliveries = client.get_database('deliveries').get_collection(...
                                                    'deliveries');

n_records = deliveries.count_documents(py.dict());
fprintf('number of records in deliveries = %d\n', n_records)

query = py.dict(pyargs('$group', py.dict(pyargs(             ...
        '_id', py.dict(pyargs('F1','$origin','F2','$dest')), ...
        'tot', py.dict(pyargs('$sum','$weight')) )) ));
pipeline = py.list();
pipeline.append(query);

for x = py.list(deliveries.aggregate(pyargs('pipeline',pipeline)))
    origin = string(x{1}{'_id'}{'F1'});
    dest   = string(x{1}{'_id'}{'F2'});
    sum_weight = x{1}{'tot'};
    fprintf('%s -> %s   %6.2f\n', origin, dest, sum_weight)
end
client.close()
```

8.6 Update

In the SQLite and PostgreSQL examples I updated those respective databases by scaling the weight of packages by 2.2 with this SQL statement:

```
update orders set weight = weight*2.2 where item = 'package';
```

The equivalent in MongoDB is considerably more complex with my setup as the version I'm running, v3.6.8, does not support multiple record updates with operators (this is supported in v4.2). My options are to write a loop over the single record update function `update_one()`, or to completely replace the records which have `item = 'package'` with new entries in Python. Looping over a single record update function sounds slow so I'll take the second approach.

Python:

```
#!/usr/bin/env python
# file: mongo_update.py
from pymongo import MongoClient
```

```python
def print_records(cursor):
    for i,x in enumerate(cursor):
        origin = x['origin']
        item = x['item']
        weight = x['weight']
        dest = x['dest']
        print(f'{i+1:3d}. {origin} {item:8s} {weight:10.6f} {dest}')

def heaviest_packages(n, db):
    packages = { "item": { "$eq": "package" } }
    return db.find(packages).sort([('weight', -1)]).limit(n)

client = MongoClient('localhost:27017')
db = client.deliveries

print('Before scaling weight of packages')
print_records(heaviest_packages(3, db.deliveries))

packages = { "item": { "$eq": "package" } }
package_records = db.deliveries.find(packages)
updated_packages = []
for record in package_records:
    new_record = record
    new_record['weight'] *= 2.2
    del new_record['_id']
    updated_packages.append(new_record)

db.deliveries.delete_many(packages)
db.deliveries.insert_many(updated_packages)

print('After scaling weight of packages')
print_records(heaviest_packages(3, db.deliveries))

client.close()
```

MATLAB:

```matlab
% file: mongo_update.m
pymongo = py.importlib.import_module('pymongo');
client = pymongo.MongoClient('localhost:27017');
deliveries = client.get_database('deliveries').get_collection('deliveries');

fprintf('Before scaling weight of packages\n')
print_records(heaviest_packages(10, deliveries))
```

```
packages = py.dict(pyargs("item",py.dict(pyargs("$eq","package"))));
%        = { "item" : { "£eq" : "packages } }
package_records = deliveries.find(packages);
updated_packages = py.list();
for record = py.list(package_records)
    new_record = record;
    new_record{1}.update(pyargs('weight', new_record{1}{'weight'} * 2.2));
    new_record{1}.pop('_id'); % == del new_record['_id']
    updated_packages.append(new_record{1});
end

deliveries.delete_many(packages);
deliveries.insert_many(updated_packages);

fprintf('After  scaling weight of packages\n')
print_records(heaviest_packages(3, deliveries))

client.close()

function print_records(cursor)
    i = 0;
    for x = py.list(cursor)
        i = i + 1;
        origin = string(x{1}{'origin'});
        item = string(x{1}{'item'});
        weight = x{1}{'weight'};
        dest = string(x{1}{'dest'});
        fprintf('%3d. %s %8s %10.6f %s\n', ...
            i, origin, item, weight, dest)
    end
end

function [cursor] = heaviest_packages(n, db)
    packages = py.dict(pyargs("item",py.dict(pyargs("$eq","package"))));
    by_weight = py.list({py.tuple({'weight', int64(-1)})});
    cursor = db.find(packages).sort(by_weight).limit(int64(n));
end
```

As a sanity check, both programs print records having the three heaviest packages before and after the update. The output from both is correct but also a bit disappointing:

```
Before scaling weight of packages
  1. C30 package    5.000000 Q38
  2. Y33 package    5.000000 E99
  3. B72 package    5.000000 L20
```

```
After scaling weight of packages
  1. C30 package   11.000000 Q38
  2. Y33 package   11.000000 E99
  3. B72 package   11.000000 L20
```

There's no variation in the weight because the fake data generator only used three significant figures for weight. Consequently the maximum value of `5.00` appears many times in 100,000 records. A more convincing test is to work with a small database (for example, set `n_rows` to 50 in `fake_data.m`) then print all records before and after.

Chapter 9

Work with Redis in MATLAB Using the redis-py Python Module

2022/08/05

Redis, the “remote dictionary server”, is a fast network-based in-memory database for key/value pairs. In addition to high performance and ability to store structured values, Redis implements a publish/subscribe service so that applications can be notified of changes to keys they subscribe to. The MATLAB Production Server has a Redis interface. Python does too—several, actually. In this article I’ll demonstrate a MATLAB interface to Redis via the the `redis-py` Python module.

9.1 Why would a MATLAB user care about Redis?

Redis may prove useful to you if you have a MATLAB application that needs to share values rapidly with other programs on your network. The programs involved use Redis as a remote memory bank for shared variables. Interaction with a Redis server happens with the Redis communication API which is available for more than 50 languages including Python—and therefore, by extension, MATLAB as well. While that sounds complicated to set up, it is often easier than writing other networking code, whether directly with TCP/IP sockets, or with a networking framework such as MPI. It is certainly cleaner and faster than communicating through the file system.

Note: Redis stores all values as character (or byte) arrays. For this reason it is poorly-suited for exchanging numeric arrays as might be needed by a computationally intensive parallel MATLAB program. Stick with MPI from the Parallel Computing Toolbox for such applications (or use Dask with MATLAB as shown in Chapter 14).

If your MATLAB projects work in isolation, that is, they don’t need to get or provide information to other applications, chances are you won’t need a network-based key/value store. If that’s the case, the rest of this article may not interest you.

9.2 Install `redis-py`

You'll need the `redis-py` Python module in your installation to call it from MATLAB.

> **NOTE:** There's a confusing overlap of names for the Python Redis package, the Redis application, and Anaconda's Redis bundle.

Is it `redis` or `redis-py` ?

- The primary Python Redis module is developed at https://github.com/redis/redis-py . Although the name on Github is `redis-py`, once you install this module, you import it with `import redis` (not `import redis-py`)
- In Anaconda distributions, this Python module is installed with `conda install redis-py` (not `conda install redis`)
- In Anaconda distributions, the full-up Redis application including the server and command line tools can be installed with `conda install redis`
- In other Python distributions, the module can be installed with `python -m pip install redis` (not `python -m pip install redis-py`)

Check your installation

Check your Python installation to see if it has the `redis-py` Python module by starting the Python command line REPL and importing it—with `import redis`, not `import redis-py`. If it imports successfully you can also check the version number.

```
> python
>>> import redis
>>> redis.__version__
'3.5.3'
```

If you get a `ModuleNotFoundError`, you'll need to install it using one of the methods listed above.

9.3 Start a Redis server

Skip this section if you already have a Redis server running somewhere on your network (or on your own computer).

To begin, install the Redis application by following the Redis installation instructions, using your operating system's package manager (for example on Ubuntu you'd need `apt install redis-tools redis-server`) or, if you have the Anaconda Python installation, by installing Redis with conda:

```
> conda install redis
```

This provides both the server, `redis-server` and command line tools such as the `redis-cli` client program. Next, open a terminal and start the server:

```
> redis-server  redis.conf --port 12987
```

The configuration file `redis.conf` and port value `--port 12987` are optional. If you don't specify them the server will use a default configuration and run on port 6379. You'll need to create a config file if you want to add password protection and persist the memory database to disk, among many other options.

If all is well you'll see some ASCII art:

```
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 5.0.3 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 12987
 |    `-._   `._    /     _.-'    |     PID: 2453308
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'

2453308:M 30 Jul 2022 14:30:20.334 # Server initialized
2453308:M 30 Jul 2022 14:30:20.334 * Ready to accept connections
```

9.4 Connect to a Redis server

We'll use the `redis-cli` command line application to test our ability to interact with the Redis server. It starts a read-evaluate-print loop (REPL) that accepts Redis commands:

```
(matpy) > redis-cli -h localhost -p 12987
localhost:12987>
```

The prompt above indicates a successful connection to the server running on the same computer. Had the connection failed, perhaps because the server isn't running, the prompt would look like this:

```
(matpy) > redis-cli -h localhost -p 12987
Could not connect to Redis at localhost:12987: Connection refused
not connected>
```

The `-h localhost` switch is redundant as the default host is the current computer. It appears here explicitly to give a sense of how one would connect to a Redis server on a different computer. Also the `-p 12987` can be omitted if your Redis server is using the default port (6379).

9.4.1 Add or update a key/value pair

Within a successfully-connected `redis-cli` session we can add new keys and values with the SET command:

```
localhost:12987> set sensor_T712_temperature 23.4
OK
localhost:12987>
```

Any Redis-capable application on our network can now retrieve the value for `sensor_T712_temperature`. To update the value, simply `set` it again:

```
localhost:12987> set sensor_T712_temperature 37.9
OK
localhost:12987>
```

9.4.2 Get a key's value

We can retrieve a key's value with the GET command

```
localhost:12987> get sensor_T712_temperature
"37.9"
localhost:12987>
```

Note that the return value is a character array, not a floating point number.

9.4.3 Batch `redis-cli` commands

Exit the `redis-cli` REPL session above with <*ctrl*>-c, <*ctrl*>-d, or by enterring either `exit` or `quit`. The command line client program accepts complete Redis commands without needing to entering the REPL. This makes it possible to write Redis-capable shell or Windows `.bat` scripts, or in fact be used by any programming language capable of making a system call. We'll start with the `ping` command which merely tests the connection. A reply of `PONG` indicates success:

```
> redis-cli -h localhost -p 12987 ping
PONG

> redis-cli -h localhost -p 12987 set sensor_T712_temperature 40.2
OK

> redis-cli -h localhost -p 12987 get sensor_T712_temperature
"40.2"
```

Now that we've checked that our Redis server is up and we can interact with it, let's do the interactions programmatically:

Python:

```python
#!/usr/bin/env python
# file: set_get.py
import redis
R = redis.Redis(host='localhost',port=12987,decode_responses=True)

try:
    R.ping()
except redis.exceptions.ConnectionError as e:
    print(f'Is Redis running?  Unable to connect {e}')
    sys.exit(1)
print(f'Connected to server.')

R.set('sensor_T712_temperature', 39.8)
retrieved_temp = R.get('sensor_T712_temperature')
print(f'sensor_T712_temperature = {retrieved_temp}')
print(f'(as a number)           = {float(retrieved_temp):.3f}')
```

A note about the `decode_responses=True` option: without it, programmatic reads of Redis keys and values receive *byte arrays*. In most cases you'd rather have strings, not byte arrays. You can get a string from a byte array by applying the `.decode()` method to the byte array, but including the `decode_responses=True` in the initial connection call spares you from having to add `.decode()` to every Redis return variable.

In any event, you still have to explicitly typecast the string temperature value if you want to use it numerically.

Running the Python program gives:

```
> ./set_get.py
Connected to server.
sensor_T712_temperature = 39.8
(as a number)           = 39.800
```

Now in MATLAB:

MATLAB:

```
% file: set_get.m
redis = py.importlib.import_module('redis');
R = redis.Redis(pyargs('host','localhost','port',int64(12987),...
                       'decode_responses',py.True));
try
  R.ping();
  fprintf('Connected to server.\n')
catch EO
  fprintf('Is Redis running?  Unable to connect: %s\n', EO.message)
end

R.set('sensor_T712_temperature', 39.8);
retrieved_temp = R.get('sensor_T712_temperature');
fprintf('sensor_T712_temperature = %s\n', string(retrieved_temp))
as_number = double(string(retrieved_temp));
fprintf('(as a number)           = %6.3f\n', as_number)
```

```
>> set_get
Connected to server.
sensor_T712_temperature = 39.8
(as a number)           = 39.800
```

9.5 Set/get structured data

The ‘value’ part of a Redis key/value pair may be a structure. Supported structures are a string, set, sorted set, list, hash, bitmap, bitfield, hyperloglog, stream, and geospatial index. The example below demonstrates setting and getting a hash since conceptually this structure resembles a MATLAB `struct`.

9.5.1 Add a hash from the command line

The Redis command HSET adds a hash. This command

```
> redis-cli -h localhost -p 12987 hset newmark alpha 0.25 beta 0.5 gamma 'is unset'
```

adds a key `newmark` with fields `alpha`, `beta`, and `gamma` corresponding to a struct like this in MATLAB:

```
>> newmark
  struct with fields:
    alpha: 0.2500
     beta: 0.5000
    gamma: "is unset"
```

Bear in mind though the numeric values in the MATLAB struct are saved as strings in the Redis hash. The Redis hash can be retrieved in its entirety with HGETALL

```
> redis-cli -h localhost -p 12987 hgetall newmark
1) "alpha"
2) "0.25"
3) "beta"
4) "0.5"
5) "gamma"
6) "is unset"
```

or selectively with HMGET (where `M` is for "multi-values") by providing a list of the desired fields:

```
> redis-cli -h localhost -p 12987 hmget newmark alpha gamma
1) "0.25"
2) "is unset"
```

Programmatically, the `.hgetall()` function returns a Python dictionary in both Python and MATLAB. This is convenient when the values are strings but not that helpful if the values have to be typecast to numbers. In that case it's an equal amount of work to instead calling the `.hmget()` method to just get back a list of values (once again, byte arrays) and convert those.

Python:

```
#!/usr/bin/env python
# file: get_hash.py
import redis
R = redis.Redis(host='localhost',port=12987,decode_responses=True)

try:
    R.ping()
except redis.exceptions.ConnectionError as e:
    print(f'Is Redis running?  Unable to connect {e}')
    sys.exit(1)
print(f'Connected to server.')

values = R.hmget('newmark', ['alpha', 'beta', 'gamma'])
newmark = { 'alpha' : float(values[0]),
            'beta'  : float(values[1]),
            'gamma' : values[2] }
print(newmark)
```

Run:

```
> get_hash.py
Connected to server.
{'alpha': 0.25, 'beta': 0.5, 'gamma': 'is unset'}
```

MATLAB:

```
% file: get_hash.m
redis = py.importlib.import_module('redis');
```

```
R = redis.Redis(pyargs('host','localhost','port',int64(12987),...
                       'decode_responses',py.True));
try
  R.ping();
  fprintf('Connected to server.\n')
catch EO
  fprintf('Is Redis running?  Unable to connect: %s\n', EO.message)
end

values = R.hmget('newmark', {'alpha', 'beta', 'gamma'});
newmark = struct;
newmark.alpha = double(string(values{1}));
newmark.beta  = double(string(values{2}));
newmark.gamma = string(values{3});
disp(newmark)
```

Run:

```
>> get_hash
Connected to server.
    alpha: 0.2500
     beta: 0.5000
    gamma: "is unset"
```

9.6 Set/get performance

The following programs set, then get 100,000 keys with integer values from 0 to 99,999. The time to `.decode()` the returned byte array is also included for the `get` time since this operation is nearly always necessary. The MATLAB code is more than 2x slower on sets and a baffling 5x slower on gets.

Python:

```
#!/usr/bin/env python
# file: set_get_speed.py
import redis
import time
R = redis.Redis(host='localhost',port=12987)

try:
    R.ping()
except redis.exceptions.ConnectionError as e:
    print(f'Is Redis running?  Unable to connect {e}')
    sys.exit(1)
print(f'Connected to server.')

N = 100_000

T_s = time.time()
for i in range(N):
    R.set(f'K_%06{i}', i)
T_e = time.time()
print(f'{N} sets for integer values took {T_e-T_s:.3f} s')

T_s = time.time()
for i in range(N):
    v = R.get(f'K_%06{i}')
    v = v.decode()
T_e = time.time()
print(f'{N} gets for integer values took {T_e-T_s:.3f} s')
```

MATLAB:

```
% file: set_get_speed.m
redis = py.importlib.import_module('redis');
R = redis.Redis(pyargs('host','localhost','port',int64(12987),...
                       'decode_responses',py.True));
try
  R.ping();
  fprintf('Connected to server.\n')
catch EO
  fprintf('Is Redis running?  Unable to connect: %s\n', EO.message)
end

N = 100000;

tic
for i = 1:N
    key = sprintf('K_%06d', i-1);
    R.set(key, i-1);
end
fprintf('%d sets for integer values took %.3f s\n', N, toc)

tic
for i = 1:N
    key = sprintf('K_%06d', i-1);
    v = R.get(key);
```

```
end
fprintf('%d gets for integer values took %.3f s\n', N, toc)
```

Python:

```
> ./set_get_speed.py
Connected to server.
100000 sets for integer values took 8.377 s
100000 gets for integer values took 7.973 s
```

MATLAB:

```
>> set_get_speed
Connected to server.
100000 sets for integer values took 21.580 s
100000 gets for integer values took 44.525 s
```

Despite the lower performance, a MATLAB Redis `set` followed by a `get` takes less than a millisecond.

9.7 Subscribe to key changes

A Redis power-feature is its support for subscriptions. When a program subscribes to a key change, Redis notifies the program each time the key's value changes. The first step involves notifying Redis that you wish to monitor "keyspace events", namely, changes to keys as well as key expiration and removal. The most common type of keyspace configuration option is "KEA". It is set from the command line like so:

```
redis-cli -h localhost -p 12987 config set notify-keyspace-events KEA
```

or programmatically like this in Python:

```
# Python
import redis
R = redis.Redis(host='localhost',port=12987,decode_responses=True)
R.config_set('notify-keyspace-events', 'KEA')
```

or this MATLAB:

```
redis = py.importlib.import_module('redis');
R = redis.Redis(pyargs('host','localhost','port',int64(12987),...
                      'decode_responses',py.True));
R.config_set('notify-keyspace-events', 'KEA')
```

The next step is to define a string pattern that matches one or more keys to watch for. For this example, say our Redis instance has a collection of keys storing temperature, humidity, and light data from an array of sensors. Their names might be

```
sensor_T712_temperature
sensor_T714_temperature
sensor_T716_illumination
sensor_T712_humidity
```

To have our application to act on changes in any of the temperature keys, our subscription would look like this:

```
Sub = R.pubsub()
Sub.psubscribe('__keyspace@0__:sensor_*_temperature')
```

After setting up the subscription, the code waits for notifications. The most direct way of doing this is in a loop that sleeps (or does something else) when no notifications come in:

Python:

```
#!/usr/bin/env python
# file: subscribe.py
import time
import sys
import redis
R = redis.Redis(host='localhost',port=12987,decode_responses=True)

try:
    R.ping()
except redis.exceptions.ConnectionError as e:
    print(f'Is Redis running?  Unable to connect {e}')
    sys.exit(1)
print(f'Connected to server.')

# modify configuration to enable keyspace notification
R.config_set('notify-keyspace-events', 'KEA')
```

```
Sub = R.pubsub()
keys_to_match = 'sensor_*_temperature'
Sub.psubscribe('__keyspace@0__:' + keys_to_match)
while True:
    try:
        message = Sub.get_message()
    except redis.exceptions.ConnectionError:
        print('lost connection to server')
        sys.exit(1)
    if message is None:
        time.sleep(0.1)
        continue
    keyname = message['channel'].replace('__keyspace@0__:','')
    if keyname == keys_to_match:
        # initial subscription value
        continue
    # gets here if any of the matching keys was updated
    value = R.get(keyname)
    print(f'{keyname} = {value}')
```

MATLAB:

```
% file: subscribe.m
redis = py.importlib.import_module('redis');
R = redis.Redis(pyargs('host','localhost','port',int64(12987),...
                       'decode_responses',py.True));
try
  R.ping();
  fprintf('Connected to server.\n')
catch EO
  fprintf('Is Redis running?  Unable to connect: %s\n', EO.message)
end

% modify configuration to enable keyspace notification
R.config_set('notify-keyspace-events', 'KEA');

Sub = R.pubsub();
keys_to_match = "sensor_*_temperature";
Sub.psubscribe("__keyspace@0__:" + keys_to_match);

while 1
   try
       message = Sub.get_message();
   catch EO
       fprintf('lost connection to server: %s\n', EO.message)
```

```
        break
    end
    if message == py.None
        pause(0.1)
        continue
    end
    keyname = replace(message{'channel'}, '__keyspace@0__:','');
    if keyname == keys_to_match
        % initial subscription value
        continue
    end
    % gets here if any of the matching keys was updated
    value = string(R.get(keyname));
    fprintf('%s = %s\n', string(keyname), value)
end
```

When these programs run, they spend most of their time sleeping (line 27 in both programs for `time.sleep(0.1)` and `pause(0.1)`). Of course the sleeps should be replaced by calls to actual work if your application has better things to do. Then, when the clients receive a notification from Redis that any of the temperature sensors was updated (line 22 in Python and line 21 in MATLAB), the `message` variable is populated with the new temperature and the print statements at lines 35 (Python) and 37 (MATLAB) are executed.

The actual duration of the sleeps—or the other work done while not listening for the key changes—is not important. Keyspace change notifications are buffered and the clients will still be informed even if the clients call `message = Sub.get_message()` long after the keys actually changed.

We can run the Python and MATLAB programs simultaneously and watch them react to command line updates of some keys. A MATLAB session responding to command line key sets such as

```
> redis-cli -p 12987
127.0.0.1:12987> set sensor_A_temperature 101.5
OK
127.0.0.1:12987> set sensor_B_temperature 101.6
OK
127.0.0.1:12987> set sensor_B_humidity 44.3
OK
127.0.0.1:12987> set sensor_C_temperature 101.7
OK
```

looks like this

```
>> subscribe
Connected to server.
sensor_A_temperature = 101.5
sensor_B_temperature = 101.6
sensor_C_temperature = 101.7
```

Note the program did not respond to a change to `sensor_B_humidity` since this key name doesn't match our pattern of `sensor_*_temperature`.

Chapter 10

Solve Global Optimization Problems in MATLAB with differential_evolution()

2022/09/03

MATLAB's `fminsearch()` function finds local minima of unbounded multivariable functions. With a lucky starting guess, it may even return the global minimum. For more rigirous searches of global minima, or control over the search domain, you'll need more powerful options such as those in the Optimization Toolbox. Alternatives to the Optimization Toolbox include global optimizers from

1. the FileExchange, specifically a differential evolution solver as used in this chapter, or
2. the `scipy.optimize` module, provided that you're willing to include Python in your MATLAB workflow (Chapter 3) and implement your objective function in Python.

The first option is the most convenient for MATLAB users as it provides a pure MATLAB solution. The second option may be necessary in corporate networks that restrict access to the Internet but include Python on the same computers that have MATLAB. This post covers both options.

10.1 Differential evolution

`scipy.optimize` has three global optimizers: `shgo()`, `dual_annealing()`, and `differential_evolution()`. Of the three, the most powerful is arguably `differential_evolution()` which implements the

Storn and Price algorithm[1] of the same name.

Briefly, the differential evolution algorithm works by starting with a population of random guesses to the solution. Guesses which yield better answers to the cost function are combined to form new guesses while poorly performing guesses are eliminated. The algorithm repeats this process until the guesses converge.

10.2 An optimization problem: shape alignment

Say you have two images of a triangular widget, where the widget's position in one image is offset from the position in the other. After some image processing, perhaps with OpenCV to identify only the corners, you're left with these two images:

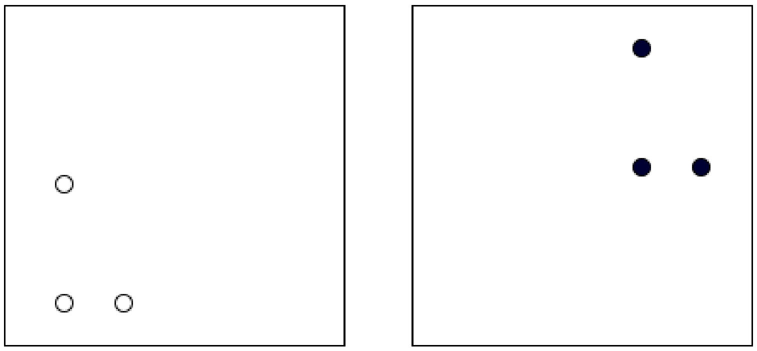


Figure 10.1: Target image on the left, known image on the right.

Our goal is to figure out how far to translate the shape on the right until it overlaps the one on the left (we'll ignore rotation and assume the widgets are always oriented the same way). Simple, right? Just compute the horizontal and vertical differences of a common corner such as the upper tip:

However if the widget on the left is on a noisy background and the corner identifcation code returns many spurious points such as shown in Figure 10.3,

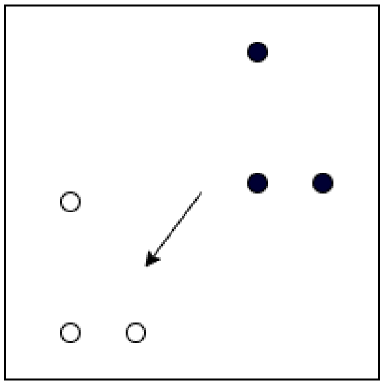


Figure 10.2: Alignment is obvious.

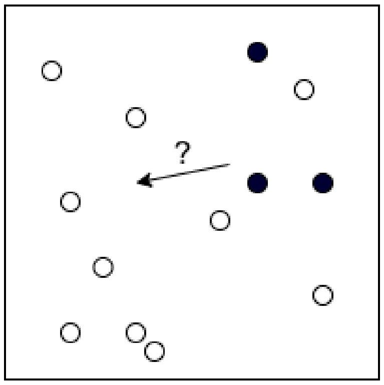


Figure 10.3: Alignment is not obvious at all (and may have multiple solutions).

then problem is much harder since there's no clear way to determine which known point corresponds to which target point.

There exist algorithms that can compute accurate offsets quickly but here we'll take a lazy approach and let an optimizer figure this out. The cost function can be the sum of distances from each corner of our known image to the nearest point on the target image. This sum should be zero if all three corners of the known image perfectly overlay the three nearest target points.

We'll start with three known points on a unit grid. To this we'll add ten target points which are made of three copies of the known points plus an offset of (-0.4, -0.2) and seven additional random points. We want an optimizer to return the offset using only the locations of the original three known points and the ten target points. The code below makes a 10 × 2 array of random coordinates for the targets then arbitrarily replaces the 3rd, 6th, and 10th points with the known

[1]Storn, R.; Price, K. (1997). "Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces". Journal of Global Optimization. 11 (4): 341–359. doi:10.1023/A:1008202821328. S2CID 5297867.

points plus the offset.

Python:

```
In : import numpy as np
In : known = np.array([[0.6, 0.5], [0.6, 0.9], [0.7,0.5]])
In : target = np.random.rand(10,2)
In : truth_offset = np.array([-0.4, -0.2])
In : target[2,:] = known[0,:] + truth_offset
In : target[5,:] = known[1,:] + truth_offset
In : target[9,:] = known[2,:] + truth_offset
```

MATLAB:

```
>> known = [0.6 0.5; 0.6 0.9; 0.7 0.5];
>> target = rand(10,2);
>> truth_offset = [-0.4 -0.2];
>> target( 3,:) = known(1,:) + truth_offset;
>> target( 6,:) = known(2,:) + truth_offset;
>> target(10,:) = known(3,:) + truth_offset;
```

Python and MATLAB will come up with different random target points which will make it harder to compare solutions. To keep things consistent I'll run the MATLAB version to produce a set of values, then hard-code these into both MATLAB and Python test environments:

Python:

```
known = np.array([[0.6, 0.5], [0.6, 0.9], [0.7,0.5]])
target = np.array([[ 8.147236863931789e-01, 1.576130816775483e-01],
                   [ 9.057919370756192e-01, 9.705927817606157e-01],
                   [ 2.0e-01               , 3.0e-01              ],
                   [ 9.133758561390194e-01, 4.853756487228412e-01],
                   [ 6.323592462254095e-01, 8.002804688888001e-01],
                   [ 2.0e-01               , 7.0e-01              ],
                   [ 2.784982188670484e-01, 4.217612826262750e-01],
                   [ 5.468815192049838e-01, 9.157355251890671e-01],
                   [ 9.575068354342976e-01, 7.922073295595544e-01],
                   [ 3.0e-01               , 3.0e-01              ]])
```

MATLAB:

```
>> known = [0.6 0.5; 0.6 0.9; 0.7 0.5];
>> target = [ 8.147236863931789e-01, 1.576130816775483e-01;
              9.057919370756192e-01, 9.705927817606157e-01;
              2.0e-01              , 3.0e-01              ;
              9.133758561390194e-01, 4.853756487228412e-01;
              6.323592462254095e-01, 8.002804688888001e-01;
              2.0e-01              , 7.0e-01              ;
              2.784982188670484e-01, 4.217612826262750e-01;
              5.468815192049838e-01, 9.157355251890671e-01;
              9.575068354342976e-01, 7.922073295595544e-01;
              3.0e-01              , 3.0e-01              ];
```

Next we need to find the closest target point for each known point To do that we'll need to compute distances from all targets to all knowns then pick the target with the smallest distance for each known point. The pairwise distance matrix computation is simple with NumPy arrays in Python and matrices in MATLAB because both support broadcasting. With broadcasting, subtracting a row vector with nR terms from a column vector with nC terms leaves us with an $nR \times nC$ matrix of all pairwise differences:

MATLAB:

```
>> format short e

>> known(:,1)'
   6.0000e-01   6.0000e-01   7.0000e-01

>> target(:,1)

   8.1472e-01
   9.0579e-01
   2.0000e-01
   9.1338e-01
   6.3236e-01
   2.0000e-01
   2.7850e-01
   5.4688e-01
   9.5751e-01
   3.0000e-01
```

then

MATLAB:

```
>> target(:,1) - known(:,1)'
```

```
  2.1472e-01   2.1472e-01   1.1472e-01
  3.0579e-01   3.0579e-01   2.0579e-01
 -4.0000e-01  -4.0000e-01  -5.0000e-01
  3.1338e-01   3.1338e-01   2.1338e-01
  3.2359e-02   3.2359e-02  -6.7641e-02
 -4.0000e-01  -4.0000e-01  -5.0000e-01
 -3.2150e-01  -3.2150e-01  -4.2150e-01
 -5.3118e-02  -5.3118e-02  -1.5312e-01
  3.5751e-01   3.5751e-01   2.5751e-01
 -3.0000e-01  -3.0000e-01  -4.0000e-01
```

The same can be done with NumPy arrays but there one must explicitly add a second dimension to one of the two column extracts with `np.newaxis`. Without this step, the arrays of x coordinates are merely one dimensional arrays with no concept of row or column orientation. (A detailed explanation can be found in section 11.1.12 of my book.)

Python:

```
In : target[:,0][:,np.newaxis] - known[:,0]
Out:
array([[ 0.21472369,  0.21472369,  0.11472369],
       [ 0.30579194,  0.30579194,  0.20579194],
       [-0.4       , -0.4       , -0.5       ],
       [ 0.31337586,  0.31337586,  0.21337586],
       [ 0.03235925,  0.03235925, -0.06764075],
       [-0.4       , -0.4       , -0.5       ],
       [-0.32150178, -0.32150178, -0.42150178],
       [-0.05311848, -0.05311848, -0.15311848],
       [ 0.35750684,  0.35750684,  0.25750684],
       [-0.3       , -0.3       , -0.4       ]])
```

By extending the broadcasting idea to include the delta y computation then squaring and summing, we can get the complete 10×3 array of squares of distances between points with just one (long) line of code:

Python:

```
In : dist2 = ( target[:,0][:,np.newaxis] - known[:,0] )**2 + \
   :         ( target[:,1][:,np.newaxis] - known[:,1] )**2
In : dist2
Out:
array([[0.16333506, 0.5972446 , 0.13039033],
       [0.31496628, 0.09849205, 0.26380789],
       [0.2       , 0.52      , 0.29      ],
       [0.0984183 , 0.27011778, 0.04574313],
       [0.09121548, 0.01099111, 0.09474363],
       [0.2       , 0.2       , 0.29      ],
       [0.10948469, 0.33207567, 0.18378505],
       [0.1756576 , 0.00306918, 0.1962813 ],
       [0.21319626, 0.1394304 , 0.15169489],
       [0.13      , 0.45      , 0.2       ]])
```

MATLAB:

```
>> dist2 = ( target(:,1) - known(:,1)' ).^2 + ...
          ( target(:,2) - known(:,2)' ).^2

   1.6334e-01   5.9724e-01   1.3039e-01
   3.1497e-01   9.8492e-02   2.6381e-01
   2.0000e-01   5.2000e-01   2.9000e-01
   9.8418e-02   2.7012e-01   4.5743e-02
   9.1215e-02   1.0991e-02   9.4744e-02
   2.0000e-01   2.0000e-01   2.9000e-01
   1.0948e-01   3.3208e-01   1.8379e-01
   1.7566e-01   3.0692e-03   1.9628e-01
   2.1320e-01   1.3943e-01   1.5169e-01
   1.3000e-01   4.5000e-01   2.0000e-01
```

meaning, for example, the square of the distance between the 8^{th} target point (== 8^{th} row) and 3^{rd} known point (== 3^{rd} column) is 1.9628e-01. All that remains is to take the square root of every term.

10.3 Setting up the cost function

Now that we know how far apart each target point is from each known point, we just need to find the closest ones then sum their distances. Taking the minumum over each column, then summing gives us the cost function:

Python:

```
In : dist  = np.sqrt( ( target[:,0][:,np.newaxis] - known[:,0] )**2 +
   :                  ( target[:,1][:,np.newaxis] - known[:,1] )**2 )
```

```
In : np.min(dist, axis=0)
Out: array([0.30201901, 0.05540018, 0.21387643])

In : np.min(dist, axis=0).sum()
Out: 0.5712956164197415
```

MATLAB:

```
>> dist = sqrt( ( target(:,1) - known(:,1)' ).^2 + ...
                ( target(:,2) - known(:,2)' ).^2 );

>> min(dist)

    0.3020    0.0554    0.2139

>> sum(min(dist))

    0.5713
```

Optimizers need to call the function that's to be minimized so we'll rewrite the expressions above as functions called `cost()`[2] We merely need to wrap the distance summation computation into a function:

Python:

```
import numpy as np
def cost(known, target):
    dist = np.sqrt( ( target[:,0][:,np.newaxis] - known[:,0] )**2 +
                    ( target[:,1][:,np.newaxis] - known[:,1] )**2 )
    return np.min(dist, axis=0).sum()
```

MATLAB:

```
function [sum_dist] = cost(known, target)
    dist = sqrt( ( target(:,1) - known(:,1)' ).^2 + ...
                 ( target(:,2) - known(:,2)' ).^2 );
    sum_dist = sum(min(dist));
end
```

[2]If you are using the Audio Toolbox, Communications Toolbox, or DSP System Toolbox you'll want to call the function something other than `cost()` since these toolboxes also have a `cost()` function.

10.4 Visualizing the cost function

It's instructive to see how the cost function varies across the search domain but this is only practical if the cost function has two or fewer independent variables and is inexpensive to compute. The cost function chosen for this post was obviously tailored to meet these criteria. I'd have liked to allow the target shape to be oriented at an arbitrary angle but then the offset would consist of three independent variables, x, y, and ϕ, and the cost function would need to be rendered as a 3D cube. That's possible but requires an advanced visualization tool that supports cutting planes.

Here are functions to display a contour plot of the cost function across the search domain. Both functions descretize the search domain of -1 <= x <= 1 and -1 <= y <= 1 into an evenly spaced grid of 100 $\times$ 100 points then compute the cost function at each of these. Afterwards we'll view the 100 $\times$ 100 cost terms on a contour plot.

Python:

```python
import numpy as np
import matplotlib.pyplot as plt
def plot_cost(known, target):
    N = 100
    [offset_x, offset_y] = np.meshgrid(np.linspace(-1,1,N),np.linspace(-1,1,N))
    C = np.zeros_like(offset_x) # cost at each offset point
    offset_target = np.zeros_like(target)
    for r in range(N):
        for c in range(N):
            offset = np.array([offset_x[r,c], offset_y[r,c]])
            offset_target = target + offset
            C[r,c] = cost(known, offset_target)
    fig, ax = plt.subplots()
    plt.contourf(offset_x, offset_y, C, levels=20)
    ax.set_aspect('equal')
    ax.set_xlim([-1,1.0])
    ax.set_xlabel('X offset')
    ax.set_ylabel('Y offset')
    ax.set_ylim([-1,1.0])
    plt.colorbar()
    plt.show()
```

MATLAB:

```matlab
function plot_cost(known, target)
    N = 100;
    [offset_x, offset_y] = meshgrid(linspace(-1,1,N), linspace(-1,1,N));
    C = zeros(size(offset_x)); % cost at each offset point
    offset_target = zeros(size(target));
    for r = 1:N
        for c = 1:N
            offset = [offset_x(r,c), offset_y(r,c)];
            offset_target = target + offset;
            C(r,c) = cost(known, offset_target);
        end
    end
    levels = 20;
    contourf(offset_x, offset_y, C, levels)
    colorbar()
end
```

For our first plot let's start with the easy case with just three target points which are offset from three known points:

Python:

```
import numpy as np
import matplotlib.pyplot as plt
known = np.array([[0.6, 0.5], [0.6, 0.9], [0.7,0.5]])
truth_offset = np.array([-0.4, -0.2])
target = known + truth_offset;
plot_cost(known, target)
```

There's a fairly distinct minimum around (0.4, 0.2). This is the amount I'd need to add to our target points to make them overlay the known points. Looking at it from the other direction, (-0.4, -0.2) was added to our known points to produce the targets. The cost function is fairly smooth with a clear minimum and should not present much challenge for gradient-based optimizers such as MATLAB's `fminsearch()`.

Now let's plot the harder case of 10 target points—3 from shifting the known points, plus the 7 noise points:

Python:

```
import numpy as np
import matplotlib.pyplot as plt
known = np.array([[0.6, 0.5], [0.6, 0.9], [0.7,0.5]])
target = np.array([[ 8.147236863931789e-01, 1.576130816775483e-01],
                   [ 9.057919370756192e-01, 9.705927817606157e-01],
                   [ 2.0e-01              , 3.0e-01              ],
```

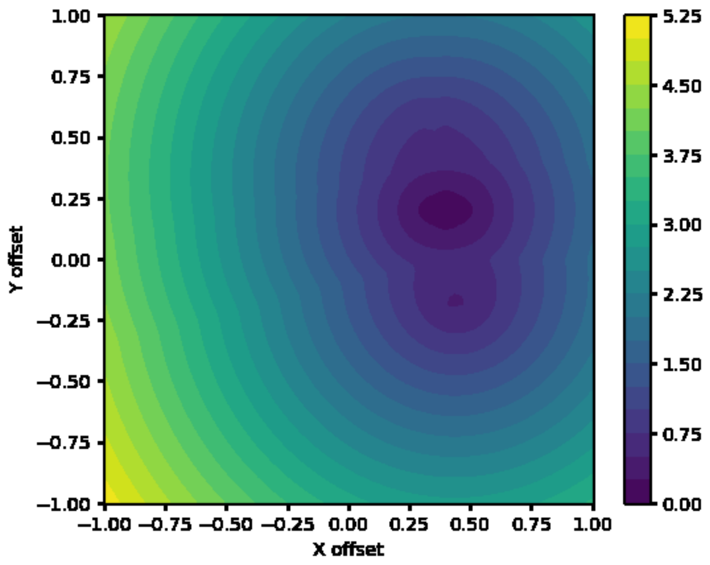


Figure 10.4: Cost function for three targets, Python.

```
                    [ 9.133758561390194e-01, 4.853756487228412e-01],
                    [ 6.323592462254095e-01, 8.002804688888001e-01],
                    [ 2.0e-01              , 7.0e-01              ],
                    [ 2.784982188670484e-01, 4.217612826262750e-01],
                    [ 5.468815192049838e-01, 9.157355251890671e-01],
                    [ 9.575068354342976e-01, 7.922073295595544e-01],
                    [ 3.0e-01              , 3.0e-01              ]])
plot_cost(known, target)
```

This time the global minimum is hard to identify. If we add even more noise points the plot will have more local minima and the optimization problem becomes even more challenging.

For completeness here are the MATLAB versions of the two plots:

MATLAB:

```
known = [0.6 0.5; 0.6 0.9; 0.7 0.5];
```

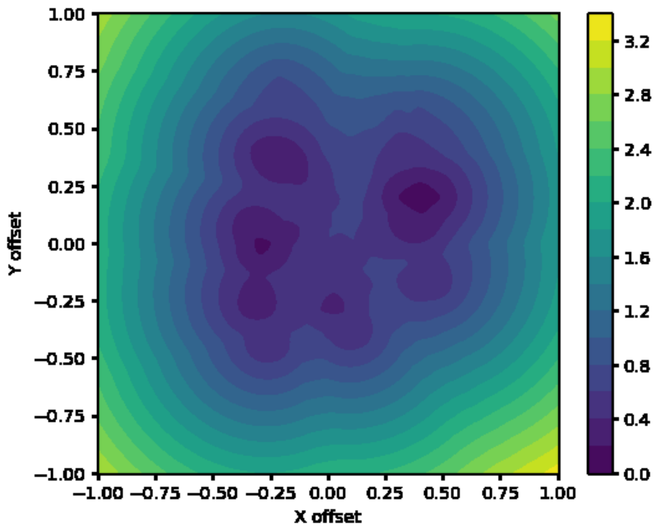


Figure 10.5: Cost function for ten targets, Python.

```
truth_offset = [-.4 -.2];
target3 = known + truth_offset;
plot_cost(known, target3)
```

MATLAB:

```
known = [0.6 0.5; 0.6 0.9; 0.7 0.5];
target = [ 8.147236863931789e-01, 1.576130816775483e-01;
           9.057919370756192e-01, 9.705927817606157e-01;
           2.0e-01              , 3.0e-01              ;
           9.133758561390194e-01, 4.853756487228412e-01;
           6.323592462254095e-01, 8.002804688888001e-01;
           2.0e-01              , 7.0e-01              ;
           2.784982188670484e-01, 4.217612826262750e-01;
           5.468815192049838e-01, 9.157355251890671e-01;
           9.575068354342976e-01, 7.922073295595544e-01;
           3.0e-01              , 3.0e-01              ];
plot_cost(known, target)
```

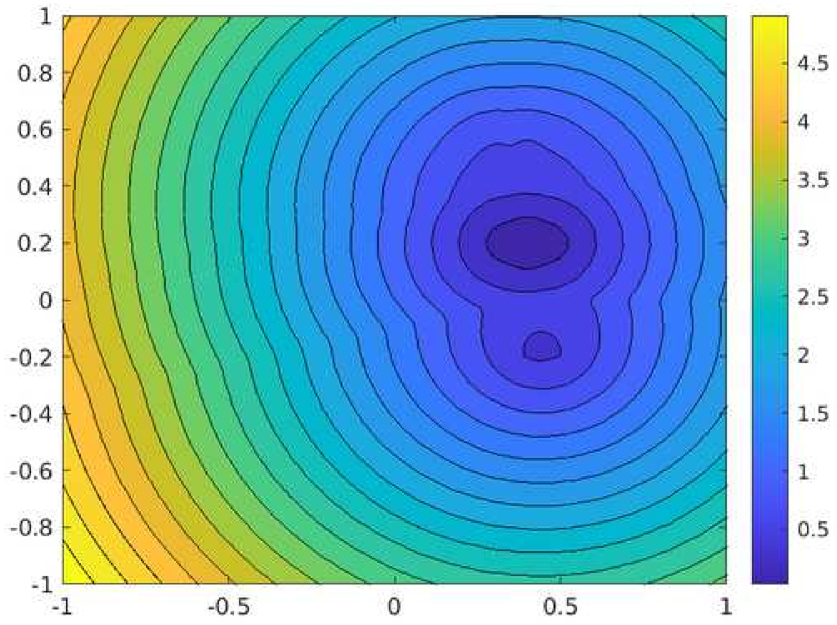


Figure 10.6: Cost function for three targets, MATLAB.

10.5 Finding the global minimum

We're ready to put our optimizers to work. Before we jump to differential evolution, let's try the `fminsearch()` function in MATLAB, first with the three target points:

MATLAB:

```
>> start_offset = [0 0];
>> fn = @(offset) cost(known, target3 + offset);
>> fminsearch(fn, start_offset)

    0.4000    0.2000
```

Looks good. Now the much harder case with the ten target points:

MATLAB:

```
>> start_offset = [0 0];
```

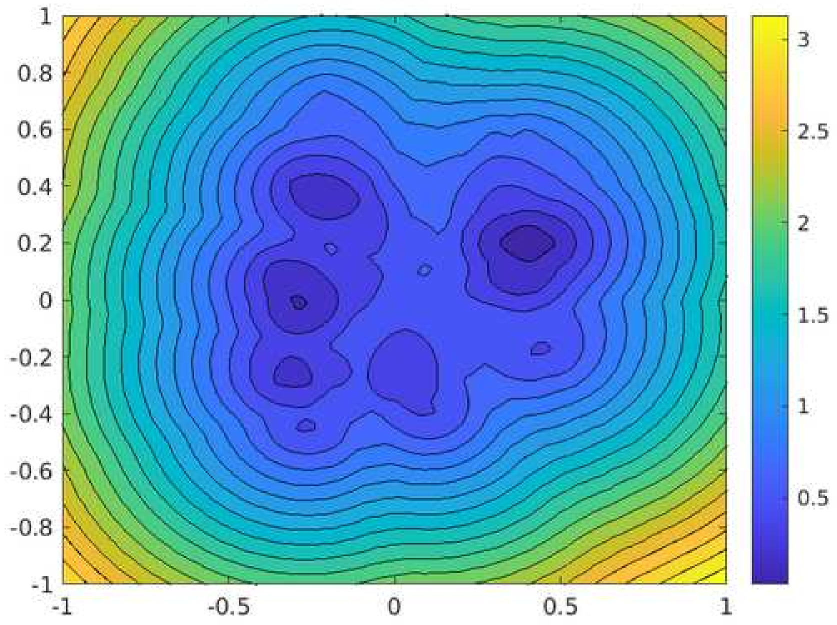


Figure 10.7: Cost function for ten targets, MATLAB.

```
>> fn = @(offset) cost(known, target + offset);
>> fminsearch(fn, start_offset)

    0.0285   -0.2725
```

Not so good—(0.0285, -0.2725) is one of the several local minima. The Optimization Toolbox has functions to find the global minimum but if that's not available, a good alternative is a differential evolution based solver from the FileExchange or from `scipy.optimize`.

Python solution with `scipy.optimize.differential_evolution()`

SciPy's `differential_evolution()` calling arguments are similar to MATLAB's `fminsearch()` but include an additional array of bounds on the independent variables (the x and y values of the offset). The first argument is a handle to a function, we'll call it `fn()` to match the `@fn()` anonymous function used in MATLAB since it serves the same purpose—in both Python and MATLAB, `fn` calls our cost function with trial values for independent variables plus other required arguments, namely the `known` and `target` arrays, passed through with the keyword argument `args`. Here's what that looks like:

Python:

```
def fn(independent_vars, *data):
    x_offset, y_offset = independent_vars
    known, target = data
    offset_target = target + np.array([x_offset, y_offset])
    return cost(known, offset_target)

bounds=([-1.0, 1.0], [ -1.0, 1.0])

result = differential_evolution(fn, bounds, args=(known, target))
```

Solutions for the three and ten point target cases return the correct global minimum for both cases:

Python:

```
In : differential_evolution(fn, bounds, args=(known, target3))
Out:
     fun: 4.052626611931048e-16
 message: 'Optimization terminated successfully.'
    nfev: 3033
     nit: 98
 success: True
       x: array([0.4, 0.2])

In : differential_evolution(fn, bounds, args=(known, target))
Out:
     fun: 3.3664710476199897e-16
 message: 'Optimization terminated successfully.'
    nfev: 2913
     nit: 94
 success: True
       x: array([0.4, 0.2])
```

Differential evolution involves random guesses for the independent variables so the number of function evaluations (`nfev` in the solution structure) varies between runs. Oddly, in this instance, the easier case of three target points required more evaluations than the harder case of ten points. If the cost function is expensive—for example a structural analysis run using Ansys or NASTRAN to find the maximum stress in a finite element model—you may want to modify the differential_evolution() optional parameters to find a suitable solution with fewer evaluations, or to run on multiple processors.

Individual elements of the returned structure are accessed through dot notation, for example,

Python:

```
In : result = differential_evolution(fn, bounds, args=(known, target3))
In : result.nfev
Out: 3213

In : result.x
Out: array([0.4, 0.2])
```

The complete Python program with all code above is `optim_w_DE.py`.

MATLAB solution with `differential_evolution()` from the FileExchange

Markus Buehren posted a thorough implementation of differential evolution entirely in MATLAB on the FileExchange. I didn't spend enough time with it to learn how to pass arbitrary inputs to the cost function so the implementation below embeds the `known` and `target` arrays directly in `cost()`.

I expanded the zip file to the directory `/home/al/matlab/DE` on my computer. To use Markus's solver this directory must be in the MATLAB search path so the script below begins with an `addpath()` to this location.

MATLAB: `optim_w_fileexchange.m`

```
% file: optim_w_fileexchange.m
addpath('/home/al/matlab/DE'); % location of unzipped files
optimInfo.title = 'Shape alignment';
tolerance = 0.0001;
paramDefCell = {
        'parameter1', [-1 1], tolerance  % offset_x
        'parameter2', [-1 1], tolerance  % offset_y
};
objFctParams.offset_x = 0;
objFctParams.offset_y = 0;
objFctHandle = @cost;
DEParams = getdefaultparams;
DEParams.maxiter = 50;
objFctSettings = 100;
emailParams = [];  % no email
[bestmem, bestval, bestFctParams, nrOfIterations, resultFileName] = ...
    differentialevolution(DEParams, paramDefCell, objFctHandle,     ...
    objFctSettings, objFctParams, emailParams, optimInfo);

function [sum_dist] = cost(scale, params)
    offset_x = params.parameter1(1);
    offset_y = params.parameter2(1);
    known = [0.6 0.5; 0.6 0.9; 0.7 0.5];
    target = [ 8.147236863931789e-01, 1.576130816775483e-01;
               9.057919370756192e-01, 9.705927817606157e-01;
               2.0e-01              , 3.0e-01              ;
               9.133758561390194e-01, 4.853756487228412e-01;
               6.323592462254095e-01, 8.002804688888001e-01;
               2.0e-01              , 7.0e-01              ;
               2.784982188670484e-01, 4.217612826262750e-01;
               5.468815192049838e-01, 9.157355251890671e-01;
               9.575068354342976e-01, 7.922073295595544e-01;
               3.0e-01              , 3.0e-01              ];
    offset_target = target + [offset_x, offset_y];
    dist = sqrt( ( offset_target(:,1) - known(:,1)' ).^2 + ...
                 ( offset_target(:,2) - known(:,2)' ).^2 );
    sum_dist = scale*sum(min(dist));
end
```

Running this script yields the correct solution

MATLAB:

```
                                        :
'Shape alignment' finished after given maximum number of 50 iterations.

Elapsed time:                   0.37 seconds
Number of function evaluations: 961
Mean function evaluation time:  0.0003928 seconds

Final result after 50 iteration(s).
Best member:
parameter1 = 0.4;
parameter2 = 0.2;
Evaluation value: 5.10014e-14

Results are saved in file cost_result_01.mat.
```

and additionally brings up plots and writes `.mat` files to the working directory.

MATLAB solution with `scipy.optimize.differential_evolution()`

If the FileExchange implementation of differential evolution isn't an option for you, use the implementation in SciPy instead. Although `scipy.optimize.differential_evolution()` can't be called from MATLAB directly—it requires a Python evaluation function, the `fn()` mentioned above—the call can be put in a Python wrapper and the wrapper called from MATLAB. This means, however, the cost function must also be implemented in Python (or I suppose could use the MATLAB engine in Python to call the MATLAB cost function from Python although I've never attempted this). Here's what the wrapper looks like for our problem:

Python: `diffev_bridge.py`

```
# file: diffev_bridge.py
import numpy as np
from scipy.optimize import differential_evolution

def cost(known, target):
    dist = np.sqrt( ( target[:,0][:,np.newaxis] - known[:,0] )**2 +
                    ( target[:,1][:,np.newaxis] - known[:,1] )**2 )
    return np.min(dist, axis=0).sum()

def fn(independent_vars, *data):
    x_offset, y_offset = independent_vars
    known, target = data
    offset_target = target + np.array([x_offset, y_offset])
    return cost(known, offset_target)

def call_DE(bounds, known, target):
    args = (known, target)
    DE_result = differential_evolution(fn, bounds, args=args)
    soln = { 'success' : DE_result.success,
             'message' : DE_result.message,
             'x' : DE_result.x,
             'nfev' : DE_result.nfev,
           }
    return soln
```

The MATLAB invocation to this wrapper is

MATLAB (>= 2020b): `optim_w_python.m`

```
np = py.importlib.import_module('numpy');
diffev = py.importlib.import_module('diffev_bridge');

known = [0.6 0.5; 0.6 0.9; 0.7 0.5];
offset = [-.4 -.2];
target = [ 8.147236863931789e-01, 1.576130816775483e-01;
           9.057919370756192e-01, 9.705927817606157e-01;
           2.0e-01              , 3.0e-01              ;
           9.133758561390194e-01, 4.853756487228412e-01;
           6.323592462254095e-01, 8.002804688888001e-01;
           2.0e-01              , 7.0e-01              ;
           2.784982188670484e-01, 4.217612826262750e-01;
           5.468815192049838e-01, 9.157355251890671e-01;
           9.575068354342976e-01, 7.922073295595544e-01;
           3.0e-01              , 3.0e-01              ];
target3 = known + offset;
bounds = py.tuple({[-1, 1], [-1, 1]});
result = diffev.call_DE(bounds, np.array(known), np.array(target3))
result = diffev.call_DE(bounds, np.array(known), np.array(target))
```

running it returns a solution dictionary for the three point and ten point cases:

MATLAB (>= 2020b):

```
>> optim_w_python
result =
  Python dict with no properties.
    {'success': True,
     'message': 'Optimization terminated successfully.',
     'x': array([0.4, 0.2]),
     'nfev': 3153}

result =
  Python dict with no properties.
    {'success': True,
     'message': 'Optimization terminated successfully.',
     'x': array([0.4, 0.2]),
     'nfev': 3033}
```

Individual items from the solution dictionary can be accessed using the dictionary `.get()` method. Alternatively, convert the dictionary to a native MATLAB `struct` with `py2mat.m`:

MATLAB (>= 2020b):

```
>> offset = double(result.get('x'))

    0.4000    0.2000

>> soln = py2mat(result)
```

```
  struct with fields:

    success: 1
    message: "Optimization terminated successfully."
          x: [0.4000 0.2000]
       nfev: 2943

>> soln.x

    0.4000    0.2000
```

Avoiding MATLAB crashes on Linux

MATLAB 2020b on Ubuntu 20.04 may crash at `diffev.call_DE()` when running `optim_w_python.m`. The solution is to start MATLAB from a suitably-configured Python environment *and* to include these lines in `startup.m`:

MATLAB (>= 2020b):

```
py.sys.setdlopenflags(int32(10));
os = py.importlib.import_module('os');
os.environ.update(pyargs('KMP_DUPLICATE_LIB_OK','TRUE'))
```

10.6 Additional Python optimization packages

Optimization is a massive topic that spans problem classes that differential evolution cannot solve. See Keivan Tafakkori's list of optimization packages for Python for an overview of additional solvers.

Chapter 11

Solve Systems of Nonlinear Equations in MATLAB with scipy.optimize.root()

2022/09/17

11.1 Introduction

Systems of nonlinear equations appear in many disciplines of science, engineering, and economics[1]. Examples include mechanical linkages, power flow, heat transfer, and Calvo pricing. MATLAB's Optimization Toolbox has functions that can solve such equations. If you don't have access to this toolbox and you're willing to include Python in your MATLAB workflow (Chapter 3), you can use SciPy's `scipy.optimize.root()` to find solutions to systems of nonlinear equations in MATLAB.

11.2 Example: The GPS Equations

Satellites in the global positioning system (GPS) constellation continuously transmit a navigation message which includes information about their orbital positions and the precise time. A GPS receiver that can get signals from at least four GPS satellites can compute its location by solving the nonlinear algebraic equations defining the intersection of four spheres, where the spheres' radii are determined by how long it took the messages to reach the receiver while traveling at the speed of light. Technically, only three spheres are needed to resolve two points in space, one of which can typ-

[1]Wolfgang Eichhorn & Winfried Gleißner, 2016. "Nonlinear Functions of Interest to Economics. Systems of Nonlinear Equations," Springer Texts in Business and Economics, in: Mathematics and Methodology for Economics, edition 1, chapter 7, pages 301-371, Springer

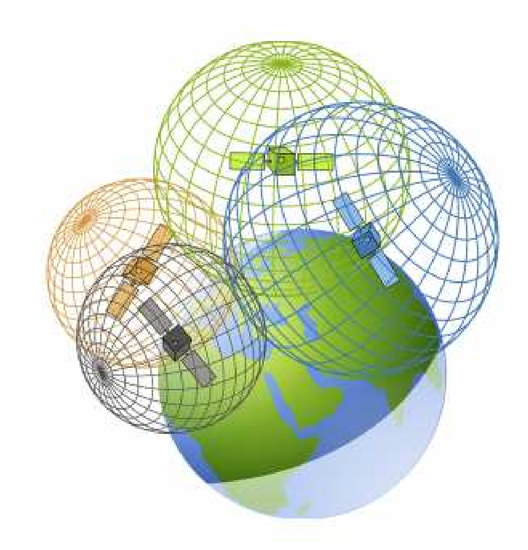


Figure 11.1: From https://gisgeography.com/trilateration-triangulation-gps/

ically be rejected as a candidate based on its distance from the earth's surface. However, since GPS receivers' clocks are inaccurate compared to the atomic clocks on the satellites, using a fourth satellite provides an extra equation that lets us additionally solve for the receiver's clock error.

The four spheres problem can actually be simplified to a system of linear equations which is much easier to solve. For the sake of this post, however, the nonlinear equations are solved directly to demonstrate how such problems are set up in Python and MATLAB.

The equations for the four spheres are

$$\begin{aligned}
(x-A_1)^2+(y-B_1)^2+(z-C_1)^2 &= (c(t_1-T_R+e))^2\\
(x-A_2)^2+(y-B_2)^2+(z-C_2)^2 &= (c(t_2-T_R+e))^2\\
(x-A_3)^2+(y-B_3)^2+(z-C_3)^2 &= (c(t_3-T_R+e))^2\\
(x-A_4)^2+(y-B_4)^2+(z-C_4)^2 &= (c(t_4-T_R+e))^2
\end{aligned}$$

where numeric subscripts denote the satellite; A, B, and C are earth-centered coordinates (such as ECI or ECEF) of each satellite; t is the time when the satellite sent its navigation message; T_R is the receiver's time; e is the error in the receiver's clock; and c is the speed of light, 299,792.458 km/s. These values come from the GMU article:

satellite	A_i [km]	B_i [km]	C_i [km]	$\Delta_i = t_i - T_R$ [seconds]
1	15600	7540	20140	0.07074
2	18760	2750	18610	0.07220
3	17610	14630	13480	0.07690
4	19170	610	18390	0.07242

We're solving for the receiver's position, $(x,\ y,\ z)$, and the error in its clock, e. Our location $(x,\ y,\ z)$ will be in the same coordinate system, ECEF or ECI, used to define the satellite positions $(A_i,\ B_i,\ C_i)$ and e will be in seconds.

11.3 Defining the function to solve

`scipy.optimize.root()`'s primary arguments are the function to solve and a starting guess. In this context, "solve" means to find values for the function's inputs so that the function returns zeros for all outputs, in other words, to find the roots of the function. To this end we'll rewrite our equations so they equal zeros then write a Python function that returns the four computed values of the left hand sides:

The input variables (x, y, z, e) need to be stored in a list or NumPy array. We'll use the Python variable `x` as a four item array so that (x, y, z, e) are stored in `x[0]` through `x[3]`.

The function also needs each satellite's position, (A_i, B_i, C_i), and the time difference between the satellite's clock and the receiver's clock, $t_i - T_R$ which we'll store as `Dt`:

Python;

```python
# file:  gps_location.py
def F(x, A, B, C, Dt):
    c_light = 299792.458 # km/s
    return (
       (x[0]-A[0])**2 + (x[1]-B[0])**2 + (x[2]-C[0])**2 - (c_light*(Dt[0] + x[3]))**2,
       (x[0]-A[1])**2 + (x[1]-B[1])**2 + (x[2]-C[1])**2 - (c_light*(Dt[1] + x[3]))**2,
       (x[0]-A[2])**2 + (x[1]-B[2])**2 + (x[2]-C[2])**2 - (c_light*(Dt[2] + x[3]))**2,
       (x[0]-A[3])**2 + (x[1]-B[3])**2 + (x[2]-C[3])**2 - (c_light*(Dt[3] + x[3]))**2,
    )
```

We'll also call this function from MATLAB so it needs to be in a file we can import. The examples below, both Python and MATLAB, will use `F()` from the file `gps_location.py`.

11.4 GPS Equations Solved in Python

We can solve for our position and clock error by populating arrays with satellite data then calling SciPy's multivariate root finder with a starting guess. The earth's radius is about 6378 km so a reasonable guess would be a random point near the earth's surface, for example (6000, 0, 0). The clock error should be much less than a second so we'll use 0 as a guess for that term.

The Python function we're finding roots of, `F()`, takes four additional arguments, namely the three arrays of satellite position, and time deltas, in addition to the input vector `x`. These additional arguments can be passed through to `F()` by giving them to `scipy.optimize.root()` via the `args` keyword option:

Python;

```
#!/usr/bin/env python
import numpy as np
import scipy.optimize
from gps_function import F

A  = np.array([15600, 18760, 17610, 19170.0]) # km/s
B  = np.array([ 7540,  2750, 14630,   610.0]) # km/s
C  = np.array([20140, 18610, 13480, 18390.0]) # km/s
Dt = np.array([0.07074, 0.07220, 0.07690, 0.07242]) # s

guess = [6000, 0, 0, 0]

location = scipy.optimize.root(F, guess, args=(A, B, C, Dt))
print(location)
```

The output includes a text string indicating whether or not the solution converged (`.message`), the number of times the function was evaluated (`.nfev`), the 4×4 Jacobian matrix (`.fjac`) and the 4 terms of function at the computed solution (`.fun`). The solution itself is in the return variable's `.x` attribute:

```
    fjac: array([[-0.42964552, -0.5286645 , -0.49262911, -0.54151189],
       [ 0.20220688, -0.32707893,  0.75636015, -0.52919852],
       [ 0.8684367 , -0.05711611, -0.42770859, -0.24417374],
       [-0.14260014,  0.78119842, -0.0479713 , -0.60588199]])
     fun: array([-1.19209290e-07, -5.96046448e-08,
                                  -1.19209290e-07, -5.96046448e-08])
 message: 'The solution converged.'
    nfev: 14
     qtf: array([ 3.35605879e-05, -1.62470220e-06,
                               -2.25947450e-06,  2.73577863e-07])
       r: array([ 6.38261526e+04,  2.49011922e+04,
                                 3.81328894e+04,  2.78876134e+10,
       -2.27583853e+04,  4.74674593e+03,
                               -1.98003469e+09, -1.01001959e+04,
       -1.33002639e+09,  2.38095380e+08])
  status: 1
 success: True
       x: array([-4.17727096e+01, -1.67891941e+01,
                        6.37005956e+03,  3.20156583e-03])
```

```
(-1.1920928955078125e-07, -5.960464477539063e-08,
            -1.1920928955078125e-07, -5.960464477539063e-08)
```

From `location.x` we see that our GPS receiver is near the North Pole at (-41.772, -16.789, 6370.060) km and its clock is off by 3.20156583e-03 seconds. `location.fun` has the function's value at the solution. Since all values should be zero, this array represents the solution's error.

11.5 GPS Equations Solved in MATLAB

MATLAB can call `scipy.optimize.root()` as easily as Python can, provided that the function we're finding the roots of is implemented in Python. (This is the same restriction we encountered minimizing a function in MATLAB with `scipy.optimize.differential_evolution()` (section 10.3). In our case we'll reuse the function `F()` from `gps_function.py`:

MATLAB 2020b+:

```
% file:  solve_gps.m
so  = py.importlib.import_module('scipy.optimize');
gps = py.importlib.import_module('gps_function');

A  = [15600, 18760, 17610, 19170.0]; % km/s
B  = [ 7540,  2750, 14630,   610.0]; % km/s
C  = [20140, 18610, 13480, 18390.0]; % km/s
Dt = [0.07074, 0.07220, 0.07690, 0.07242]; % s

guess = [6000, 0, 0, 0];
ABCDt = py.tuple({A, B, C, Dt});

location = so.root(gps.F, guess, pyargs('args',ABCDt))
```

The `location` found in MATLAB is the same as the one found in Python:

```
>> solve_gps

location =

  Python OptimizeResult with no properties.

      fjac: array([[-0.42964552, -0.5286645 , -0.49262911, -0.54151189],
         [ 0.20220688, -0.32707893,  0.75636015, -0.52919852],
         [ 0.8684367 , -0.05711611, -0.42770859, -0.24417374],
         [-0.14260014,  0.78119842, -0.0479713 , -0.60588199]])
       fun: array([-1.19209290e-07, -5.96046448e-08,
                           -1.19209290e-07, -5.96046448e-08])
   message: 'The solution converged.'
      nfev: 14
       qtf: array([ 3.35605879e-05, -1.62470220e-06,
                           -2.25947450e-06,  2.73577863e-07])
         r: array([ 6.38261526e+04,  2.49011922e+04,
                              3.81328894e+04,  2.78876134e+10,
         -2.27583853e+04,  4.74674593e+03,
                             -1.98003469e+09, -1.01001959e+04,
         -1.33002639e+09,  2.38095380e+08])
    status: 1
   success: True
         x: array([-4.17727096e+01, -1.67891941e+01,
                              6.37005956e+03,  3.20156583e-03])
```

Note the object's class: `Python OptimizeResult`. Extracting values from such an object is possible but clumsy as the return values are also Python objects:

```
>> format long e
>> location.get('x')

ans =

  Python ndarray:

    -4.177270957085472e+01    -1.678919410652270e+01
               6.370059559223343e+03     3.201565829594195e-03

    Use details function to view the properties of the Python object.

    Use double function to convert to a MATLAB array.
```

We'd much prefer a native MATLAB struct. That's easily done with `py2mat.m`[2]

```
>> loc = py2mat(location)

  struct with fields:

          x: [-4.177270957085472e+01 -1.678919410652270e+01
```

[2]I added support for `Python OptimizeResult` objects in `py2mat.m` on 2022-09-16. Earlier versions will fail to convert such objects.

```
                       6.370059559223343e+03 3.201565829594195e-03]
success: 1
 status: 1
   nfev: 14
   fjac: [4×4 double]
      r: [6.382615261849605e+04 2.490119219503466e+04
                    3.813288936837602e+04 2.788761343528719e+10 ... ]
    qtf: [3.356058790625163e-05 -1.624702195475986e-06
                   -2.259474496258976e-06 2.735778627939572e-07]
    fun: [-1.192092895507812e-07 -5.960464477539062e-08
                   -1.192092895507812e-07 -5.960464477539062e-08]
message: "The solution converged."
```

Chapter 12

Symbolic Math in MATLAB with SymPy

2022/10/01

SymPy is a Python-based computer algebra system with capabilities similar to Maple, Maxima, MATLAB's Symbolic Math Toolbox, SageMath, and to a lesser extent, Mathematica. The most natural way to work with mathematical expressions symbolically in MATLAB is with the Symbolic Math Toolbox. If you don't have access to this toolbox and you're willing to include Python in your MATLAB workflow (Chapter 3), SymPy may be an adequate alternative. SymPy is included with the recommended Anaconda Python distribution.

SymPy's capabilities are vast—the PDF version of its recent v1.11.1 documentation is a staggering 3,184 pages—and this post merely covers the basics: defining symbolic variables, integrating, taking derivatives, solving simultaneous equations, summing series, generating LaTeX markup, and generating source code in MATLAB and other languages.

To give you a sense of SymPy's depth, here are a few of its capabilities that *aren't* covered in this article: linear, time-invariant (LTI) controls, combinatorics, quaternions, sets, tensor products and contraction, Pauli algebra, hydrogen wavefunctions, quantum harmonic oscillators, gamma matrices, Gaussian optics, and continuum mechanics.

12.1 Defining symbolic variables and functions

The first steps to using SymPy are importing its various modules and defining symbolic variables and functions. Symbolic variables can be defined three ways: 1) imported from `sympy.abc`, 2) created with a call to `sympy.symbols()`, or 3) created with a call to `sympy.Symbol()`. The module `sympy.abc` has a collection of predefined symbolic variables that include all lower- and uppercase letters and words defining Greek letters such as `omega`. Multi-character variables can be defined with the `sympy.symbols()` or `sympy.Symbol()` functions. The latter allows one to add properties to a variable indicating, for example, that it is an integer, or its value is always positive, or that it is a constant, or must be real. The code below defines symbolic variables `x`, `y`, `r`, `phi`, `u1`, `v1`, and `Length`:

Python:

```
from sympy import symbols, Symbol
from sympy.abc import x, y, r, phi                                  # method 1
u1, v1 = symbols('u1 v1')                                           # method 2
Length = Symbol('Length', positive=True, constant=True, real=True) # method 3
```

The constants *pi*, i, e, and ∞ must be imported to be used symbolically. i and e exist as uppercase `I` and `E`, and ∞ is `oo`:

Python:

```
In : from sympy import Float, pi, I, E, oo
In : pi, I, E, oo
Out: (pi, I, E, oo)
In : Float(pi), Float(E), Float(oo)
Out: (3.14159265358979, 2.71828182845905, oo)
```

`Float(I)` doesn't appear above because yields a `TypeError`.

Unlike π and e, infinity is not expanded to its IEEE 754 floating point representation even though NumPy has an explicit term for it: `np.inf`.

Symbolic functions

In addition to symbolic variables, functions that operate on symbolic variables must also be explicitly requested from SymPy. Symbolic-capable functions include forward and inverse trignometric, forward and inverse hyperbolic, logarithmic, and many others.

For example if your work involves symbolic computations with sine, cosine, and natural logarithm functions, load the symbolic versions of these functions from SymPy with:

Python:

```
from sympy import sin, cos, ln
```

Examples of equivalent MATLAB imports appear in the integration and derivation sections below.

12.2 Solving equations

We'll use the symbols defined above in some equations and have SymPy solve them. Here's an easy one to start: given the equation for a circle,

$$x^2 + y^2 = r^2$$

express the circle in terms of x. There are two ways to set up the problem for SymPy. The first involves recasting the equation to equal zero, as in

$$x^2 + y^2 - r^2 = 0$$

then supplying the left hand side to `solve()`, along with the variable we want to solve for, namely `x`:

Python:

```
In : from sympy.abc import x, y, r
In : from sympy import solve
In : solve(x**2 + y**2 - r**2, x)
Out: [-sqrt((r - y)*(r + y)), sqrt((r - y)*(r + y))]
```

While the answer is correct, it doesn't resemble the expected form of

$$x = \pm\sqrt{r^2 - y^2}$$

We can get the expected form by explicitly expanding the two parts of the solution:

Python:

```
In : from sympy.abc import x, y, r
In : from sympy import solve, expand
In : sol_minus, sol_plus = solve(x**2 + y**2 - r**2, x)
In : expand(sol_minus), expand(sol_plus)
Out: (-sqrt(r**2 - y**2), sqrt(r**2 - y**2))
```

... which raises the question of what does it mean for an equation to be "simplified" or "expanded"? SymPy's opinion in this case is the opposite of my understanding of these terms.

The second way to solve an equation is to create an explicit equation object with the `Eq()` function then pass the equation to `solve()`. `Eq()` takes two input arguments which are the left and right sides of the equation:

Python:

```
In : from sympy.abc import x, y, r
In : from sympy import Eq
In : circle_eqn = Eq(x**2 + y**2, r**2) # x^2 + y^2 = r^2
In : solve(circle_eqn, x)
Out: [-sqrt((r - y)*(r + y)), sqrt((r - y)*(r + y))]
```

Intersection of a circle and line in Python

So far we haven't really solved anything, we've just reordered terms of an equation. Next we'll add the equation of a line to the mix and have SymPy figure out the points of intersection. The equations are

$$\begin{aligned} x^2 + y^2 &= r^2 \\ y &= mx + b \end{aligned}$$

This is a nonlinear system of equations so we'll need SymPy's `nonlinsolve()` function:

Python:

```
In : from sympy import Eq, nonlinsolve
In : from sympy.abc import x, y, r, m, b
In : circle_eqn = Eq(x**2 + y**2, r**2) # x^2 + y^2 = r^2
In : line_eqn = Eq(y, m*x + b)          # y = m x + b
In : nonlinsolve([circle_eqn, line_eqn], [x, y])
Out: {((-b + b/(m**2 + 1) - m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1))/m,
        b/(m**2 + 1) - m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1)),
      ((-b + b/(m**2 + 1) + m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1))/m,
        b/(m**2 + 1) + m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1))}
```

SymPy returns two pairs of (x, y) values representing the points of intersection—if these exist, of course. The actual values of b, m, and r will determine whether the solution has real or imaginary values.

In the previous chapter I showed how SciPy can find numeric solutions to systems of nonlinear equations. How does SymPy fare on the intersection of four spheres problem described there? Unfortunately, not well. The program below assembles the problem but `nonlinearsolve()` doesn't return after running for more than an hour.

Python:

```
#!/usr/bin/env python
# file: four_spheres.py
# Don't try this at home; the nonlinsolve() step hangs.

from sympy import Eq, nonlinsolve, symbols
from sympy.abc import x, y, z, c, e
A1, A2, A3, A4 = symbols('A1 A2 A3 A4')
B1, B2, B3, B4 = symbols('B1 B2 B3 B4')
C1, C2, C3, C4 = symbols('C1 C2 C3 C4')
Dt1, Dt2, Dt3, Dt4 = symbols('Dt1 Dt2 Dt3 Dt4')
s1 = Eq( (x-A1)**2 + (y-B1)**2 + (z-C1)**2 , (c*(Dt1 + e) )**2 )
s2 = Eq( (x-A2)**2 + (y-B2)**2 + (z-C2)**2 , (c*(Dt2 + e) )**2 )
s3 = Eq( (x-A3)**2 + (y-B3)**2 + (z-C3)**2 , (c*(Dt3 + e) )**2 )
s4 = Eq( (x-A4)**2 + (y-B4)**2 + (z-C4)**2 , (c*(Dt4 + e) )**2 )
soln = nonlinsolve([s1, s2, s3, s4], [x, y, z, e])  # hangs
```

Intersection of a circle and line in MATLAB

The MATLAB version of the circle/line intersection problem looks like this:

MATLAB (2020b and newer):

```
sympy = py.importlib.import_module('sympy');
abc = py.importlib.import_module('sympy.abc');
x = abc.x;
y = abc.y;
r = abc.r;
m = abc.m;
b = abc.b;
Two = int64(2);
circle_eqn = sympy.Eq(x^Two + y^Two, r^Two); % x^2 + y^2 = r^2
line_eqn   = sympy.Eq(y, m*x + b);           % y = m*x + b
soln = sympy.nonlinsolve(py.list({circle_eqn, line_eqn}), ...
                         py.list({x, y}))
```

> ***Note:*** In MATLAB, exponents for squares, cubes, and so on should be explicitly cast to integers in the SymPy expression. If this is not done, the notation "^2" given to SymPy will use the default MATLAB type of `double` which greatly complicates — and often defeats — the SymPy evaluator.

The solution in MATLAB appears with the full complement of attributes in addition to just the equations we're after. This is a truncated portion of the output:

MATLAB:

```
>> soln

  Python FiniteSet with properties:

                     args: [1x2 py.tuple]
             assumptions0: [1x1 py.dict]
                 boundary: [1x1 py.sympy.sets.sets.FiniteSet]
                                :
        is_transcendental: [1x1 py.NoneType]
                  is_zero: [1x1 py.NoneType]
                  measure: [1x1 py.int]

    {((-b + b/(m**2 + 1) - m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1))/m,
        b/(m**2 + 1) - m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1)),
     ((-b + b/(m**2 + 1) + m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1))/m,
        b/(m**2 + 1) + m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1))}
```

The string containing just the two (x, y) pairs is in `soln.char`:

MATLAB:

```
>> soln.char

    '{((-b + b/(m**2 + 1) - m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1))/m,
         b/(m**2 + 1) - m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1)),
      ((-b + b/(m**2 + 1) + m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1))/m,
         b/(m**2 + 1) + m*sqrt(-b**2 + m**2*r**2 + r**2)/(m**2 + 1))}'
```

12.3 Solution as LaTeX

The SymPy text results we've seen so far look awkward. If you want to publish the results they'll look much more elegant rendered with LaTeX. Simple expressions are easy to write manually but SymPy's output can get lengthy so its LaTeX generator comes in handy:

Python:

```
In : from sympy.printing.latex import print_latex
In : from sympy import Eq, nonlinsolve
In : from sympy.abc import x, y, r, m, b
In : circle_eqn = Eq(x**2 + y**2, r**2) # x^2 + y^2 = r^2
In : line_eqn = Eq(y, m*x + b)          # y = m x + b
In : soln = nonlinsolve([circle_eqn, line_eqn], [x, y])
In : print_latex(soln)
```

The last line produces this output

```
\left\{\left( \frac{- b + \frac{b}{m^{2} + 1} - \frac{m \sqrt{- b^{2} + m^{2} r^{2}
+ r^{2}}}{m^{2} + 1}}{m}, \  \frac{b}{m^{2} + 1} - \frac{m \sqrt{- b^{2} + m^{2} r^{2}
+ r^{2}}}{m^{2} + 1}\right), \left( \frac{- b + \frac{b}{m^{2} + 1} + \frac{m \sqrt{-
b^{2} + m^{2} r^{2} + r^{2}}} {m^{2} + 1 }}{m}, \  \frac{b}{m^{2} + 1} + \frac{m \sqrt{-
b^{2} + m^{2} r^{2} + r^{2}}}{m^{2} + 1}\right)\right\}
```

which renders as

$$\left(\frac{- b + \frac{b}{m^{2} + 1} - \frac{m \sqrt{- b^{2} + m^{2} r^{2} + r^{2}}}{m^{2} + 1}}{m}, \ \frac{b}{m^{2} + 1} - \frac{m \sqrt{- b^{2} + m^{2} r^{2} + r^{2}}}{m^{2} + 1}\right),$$
$$\left(\frac{- b + \frac{b}{m^{2} + 1} + \frac{m \sqrt{- b^{2} + m^{2} r^{2} + r^{2}}}{m^{2} + 1}}{m}, \ \frac{b}{m^{2} + 1} + \frac{m \sqrt{- b^{2} + m^{2} r^{2} + r^{2}}}{m^{2} + 1}\right)$$

12.4 Solution as MATLAB / Python / C / Fortran / etc source code

In addition to LaTeX, SymPy can express solutions as computer source code in a dozen languages. The `sympy.printing` module has these emitters:

Language	`sympy.printing` function
C	`ccode()`
C++	`cxxcode()`
Fortran	`fcode()`
GLSL	`glsl_code()`
JavaScript	`jscode()`
Julia	`julia_code()`
Maple	`maple_code()`
Mathematica	`mathematica_code()`
MATLAB	`octave_code()`
Python	`pycode()`
R	`rcode()`
Rust	`rust_code()`

The code blocks below repeat the circle/line intersection this time emitting MATLAB source code instead of LaTeX. In MATLAB, the call to `octave_code()` returns a Python string so this is converted to a MATLAB string before printing.

Python:

```
#!/usr/bin/env python
from sympy import Eq, nonlinsolve
from sympy.abc import x, y, r, m, b
from sympy.printing import octave_code
circle_eqn = Eq(x**2 + y**2, r**2) # x^2 + y^2 = r^2
line_eqn = Eq(y, m*x + b)          # y = m x + b
soln = nonlinsolve([circle_eqn, line_eqn], [x, y])
print(soln)
print(octave_code(soln))
```

MATLAB (2020b and newer):

```
sympy = py.importlib.import_module('sympy');
abc = py.importlib.import_module('sympy.abc');
sym_code = py.importlib.import_module('sympy.printing');
x = abc.x;
y = abc.y;
r = abc.r;
m = abc.m;
b = abc.b;
Two = int64(2);
circle_eqn = sympy.Eq(x^Two + y^Two, r^Two); % x^2 + y^2 = r^2
line_eqn   = sympy.Eq(y, m*x + b);           % y = m*x + b
soln = sympy.nonlinsolve(py.list({circle_eqn, line_eqn}), ...
                         py.list({x, y}));
fprintf('%s\n', string(sym_code.octave_code(soln)) )
```

Both versions produce this output:

```
"{{(-b + b./(m.^2 + 1) - m.*sqrt(-b.^2 + m.^2.*r.^2 + r.^2)./(m.^2 + 1))./m,
    b./(m.^2 + 1) - m.*sqrt(-b.^2 + m.^2.*r.^2 + r.^2)./(m.^2 + 1)},
  {(-b + b./(m.^2 + 1) + m.*sqrt(-b.^2 + m.^2.*r.^2 + r.^2)./(m.^2 + 1))./m,
    b./(m.^2 + 1) + m.*sqrt(-b.^2 + m.^2.*r.^2 + r.^2)./(m.^2 + 1)}}"
```

Replacing `octave_code()` with `jscode()` produces much lengthier JavaScript code:

```
{((-b + b/(Math.pow(m, 2) + 1) - m*Math.sqrt(-Math.pow(b, 2) +
     Math.pow(m, 2)*Math.pow(r, 2) + Math.pow(r, 2))/(Math.pow(m, 2) + 1))/m,
   b/(Math.pow(m, 2) + 1) - m*Math.sqrt(-Math.pow(b, 2) +
     Math.pow(m, 2)*Math.pow(r, 2) + Math.pow(r, 2))/(Math.pow(m, 2) + 1)),
 ((-b + b/(Math.pow(m, 2) + 1) + m*Math.sqrt(-Math.pow(b, 2) +
     Math.pow(m, 2)*Math.pow(r, 2) + Math.pow(r, 2))/(Math.pow(m, 2) + 1))/m,
   b/(Math.pow(m, 2) + 1) + m*Math.sqrt(-Math.pow(b, 2) +
     Math.pow(m, 2)*Math.pow(r, 2) + Math.pow(r, 2))/(Math.pow(m, 2) + 1))}
```

12.5 Definite and indefinite integrals

Integration is done with SymPy's `integrate()` function. Here's a tricky one:

In addition to the `integrate()` function we'll also need to import symbolic versions of the sine and cosine functions. Constraining n and x to be real, and n to be positive yields a cleaner result.

The Python and MATLAB solutions are

Python:

```python
#!/usr/bin/env python
from sympy import pi, sin, cos, integrate, Symbol
from sympy.printing.latex import print_latex
from sympy.printing import octave_code
n = Symbol('n', real=True, positive=True, integer=True)
x = Symbol('x', real=True)
soln = integrate(cos(pi*n*x)*(1 + sin(3*pi*n*x/2)**2), (x))
at_x_n = soln.evalf(subs={x : 1.5, n : 1})
print(f'solution at x=1.5, n=1 is {at_x_n}')
```

MATLAB:

```matlab
sympy = py.importlib.import_module('sympy');
printing = py.importlib.import_module('sympy.printing');
n = sympy.Symbol('n', pyargs('real',py.True, ...
                             'positive',py.True,'integer',py.True));
x = sympy.Symbol('x', pyargs('real',py.True));
Pi = sympy.pi;   % uppercase to be distinct from matlab's pi
Sin = sympy.sin; % uppercase to be distinct from matlab's sin()
Cos = sympy.cos; % uppercase to be distinct from matlab's cos()
soln = sympy.integrate(Cos(Pi*n*x)*(1 + Sin(3*Pi*n*x/2)^int64(2)), ...
                       py.tuple({x}));
fprintf('%s\n', soln.char)
printing.print_latex(soln)
```

> ***Note:*** In the MATLAB solution, SymPy versions of π and sine and cosine functions are imported with a leading uppercase character to make them distinct from MATLAB's `pi`, `sin()`, and `cos()`. If you import `pi`, `sin`, and `cos` from SymPy using the identical names they will be SymPy objects that cannot be used in conventional MATLAB expressions.

The text version of the solution, minus the obligatory constant, from Python is

```
7*sin(pi*n*x)*sin(3*pi*n*x/2)**2/(16*pi*n) +
9*sin(pi*n*x)*cos(3*pi*n*x/2)**2/(16*pi*n) + sin(pi*n*x)/(pi*n) -
3*sin(3*pi*n*x/2)*cos(pi*n*x)*cos(3*pi*n*x/2)/(8*pi*n)
```

while LaTeX version renders as:

$$\frac{7\sin\left(\pi nx\right)\sin^{2}\left(\frac{3\pi nx}{2}\right)}{16\pi n}+\frac{9\sin\left(\pi nx\right)\cos^{2}\left(\frac{3\pi nx}{2}\right)}{16\pi n}+\frac{\sin\left(\pi nx\right)}{\pi n}-\frac{3\sin\left(\frac{3\pi nx}{2}\right)\cos\left(\pi nx\right)\cos\left(\frac{3\pi nx}{2}\right)}{8\pi n}$$

The MATLAB result looks quite different

```
0.4375*sin(pi*n*x)*sin(3*pi*n*x/2)**2/(pi*n) +
0.5625*sin(pi*n*x)*cos(3*pi*n*x/2)**2/(pi*n) + 1.0*sin(pi*n*x)/(pi*n) -
0.375*sin(3*pi*n*x/2)*cos(pi*n*x)*cos(3*pi*n*x/2)/(pi*n)
```

but, as shown with numeric evaluation, both yield the same results.

Numeric evaluation of symbolic results

One way to convince ourselves that two starkly different equations are equivalent is to substitute numeric values in for the variables. SymPy equations have a `.evalf()` method to facilitate this substitution. The method accepts a dictionary of substitutions and returns a numeric value. If we append these lines to the Python code above,

Python:

```
at_x_n = soln.evalf(subs={x : 1.5, n : 1})
print(f'solution at x=1.5, n=1 is {at_x_n}')
```

we'll additionally get

```
solution at x=1.5, n=1 is -0.477464829275686
```

I've not been able to replicate `.evalf()'s subs=` expression in MATLAB, however. Instead, to compare the results of the two integrals, I generated MATLAB source code for each and substituted arrays of values and confirmed both Python and MATLAB solutions agree.

Definite integrals

Definite integrals are defined by supplying lower and upper bounds after the variable of integration. The value of the same integral shown above going from $x = 0$ to $x = \frac{1}{\pi}$ for $n = 1$, in other words

$$\int_0^{\frac{1}{\pi}} \cos(\pi x) \left(1 + \sin^2\left(\frac{3\pi x}{2}\right)\right) dx$$

is solved like this in Python and MATLAB:

Python:

```
#!/usr/bin/env python
from sympy import pi, sin, cos, integrate, Symbol, simplify
from sympy.printing.latex import print_latex
from sympy.printing import octave_code
x = Symbol('x', real=True)
soln = integrate(cos(pi*x)*(1 + sin(3*pi*x/2)**2), (x, 0, 1/pi))
print_latex(simplify(soln))
```

MATLAB:

```
sympy = py.importlib.import_module('sympy');
printing = py.importlib.import_module('sympy.printing');
x = sympy.Symbol('x', pyargs('real',py.True));
Pi = sympy.pi;   % uppercase to be distinct from matlab's pi
Sin = sympy.sin; % uppercase to be distinct from matlab's sin()
Cos = sympy.cos; % uppercase to be distinct from matlab's cos()
soln = sympy.integrate(Cos(Pi*x)*(1 + Sin(3*Pi*x/2)^int64(2)), ...
                       py.tuple({x, 0, 1/Pi}));
printing.print_latex(soln)
fprintf('%s\n', soln.evalf().char)
```

Once again, the Python and MATLAB symbolic solutions differ but both evaluate to the same number:

Python:

$$\frac{-2\sin(2) - \sin(4) + 24\sin(1)}{16\pi} = 0.380649112305801$$

MATLAB:

$$\frac{1.19584445481538}{\pi} = 0.380649112305801$$

In this case `.evalf()` works in MATLAB because there are no variables to substitute.

12.6 Derivatives

Derivatives are computed with the `diff()` function. If we start with a function

$$y = \sin^2(\pi x)e^{-ix}$$

we can get $\frac{dy}{dx}$ with

Python:

```
#!/usr/bin/env python
from sympy import I, pi, sin, exp, diff
from sympy.abc import x, y
from sympy.printing.latex import print_latex
soln = diff(sin(pi*x)**2 * exp(-I*x), x)
print_latex(soln)
```

MATLAB:

```
sympy = py.importlib.import_module('sympy');
abc = py.importlib.import_module('sympy.abc');
printing = py.importlib.import_module('sympy.printing');
x = abc.x;
y = abc.y;
I = sympy.I;
Pi = sympy.pi;   % uppercase to be distinct from matlab's pi
Sin = sympy.sin; % uppercase to be distinct from matlab's sin()
Exp = sympy.exp; % uppercase to be distinct from matlab's exp()
Two = int64(2);
soln = sympy.diff(Sin(Pi*x)^Two * Exp(-I*x), x);
fprintf('%s\n', soln.char)
printing.print_latex(soln)
```

both of which give this LaTeX representation of the solution:

$$\frac{dy}{dx} = -ie^{-ix}\sin^2(\pi x) + 2\pi e^{-ix}\sin(\pi x)\cos(\pi x)$$

Partial derivatives

`diff()` can also return partial derivatives. Say you have a complicated surface such as

$$z(x,y) = 4x^4 - 17x^3y + 3x^2y^2 - y^3 - y^4$$

and you want to compute the matrix of second order derivatives

$$\begin{bmatrix} \frac{\partial^2 z}{\partial^2 x} & \frac{\partial^2 z}{\partial x \partial y} \\ \frac{\partial^2 z}{\partial y \partial x} & \frac{\partial^2 z}{\partial^2 y} \end{bmatrix}$$

you could do it with

Python:

```python
#!/usr/bin/env python
from sympy import diff
from sympy.abc import x, y, z
from sympy.printing.latex import print_latex
z = 4*x**4 - 17*x**3*y + 3*x**2*y**2 - y**3 - y**4
dzz_dxx = diff(z, x, x)
dzz_dxy = diff(z, x, y)
dzz_dyx = diff(z, y, x)
dzz_dyy = diff(z, y, y)
print_latex(dzz_dxx)
print_latex(dzz_dxy)
print_latex(dzz_dyx)
print_latex(dzz_dyy)
```

MATLAB:

```matlab
sympy = py.importlib.import_module('sympy');
abc = py.importlib.import_module('sympy.abc');
printing = py.importlib.import_module('sympy.printing');
x = abc.x;
y = abc.y;
```

```
z = abc.z;
E2 = int64(2);
E3 = int64(3);
E4 = int64(4);
z = 4*x^E4 - 17*x^E3*y + 3*x^E2*y^E2 - y^E3 - y^E4;
dzz_dxx = diff(z, x, x);
dzz_dxy = diff(z, x, y);
dzz_dyx = diff(z, y, x);
dzz_dyy = diff(z, y, y);
printing.print_latex(dzz_dxx)
printing.print_latex(dzz_dxy)
printing.print_latex(dzz_dyx)
printing.print_latex(dzz_dyy)
```

which return different arrangements of the same result:

Python:

$$\begin{bmatrix} 6 \cdot \left(8x^2 - 17xy + y^2\right) & 3x\left(-17x + 4y\right) \\ 3x\left(-17x + 4y\right) & 6\left(x^2 - 2y^2 - y\right) \end{bmatrix}$$

MATLAB:

$$\begin{bmatrix} 48.0x^2 - 102.0xy + 6.0y^2 & x\left(-51.0x + 12.0y\right) \\ x\left(-51.0x + 12.0y\right) & 6.0x^2 - 12y^2 - 6y \end{bmatrix}$$

12.7 Summing series

SymPy has support for Fourier series, power series, series summation, and series expansion. This example shows how the closed-form summation

$$\sum_{k=1}^{N+1} \frac{e^{ik\pi}}{N^2}$$

can be found:

Python:

```
#!/usr/bin/env python
from sympy import I, Sum, exp, pi
from sympy.abc import k, N
from sympy.printing.latex import print_latex
s = Sum( exp(I*k*pi)/N**2, (k, 1, N+1))
soln = s.doit()
print_latex(soln)
```

MATLAB:

```
sympy = py.importlib.import_module('sympy');
abc = py.importlib.import_module('sympy.abc');
printing = py.importlib.import_module('sympy.printing');
k = abc.k;
N = abc.N;
I = sympy.I;
Pi  = sympy.pi;  % uppercase to be distinct from matlab's pi
Exp = sympy.exp; % uppercase to be distinct from matlab's exp()
Sum = sympy.Sum;
Two = int64(2);
s = Sum( Exp(I*k*Pi)/(N^Two), py.tuple({k, 1, N+1}));
soln = s.doit();
printing.print_latex(soln)
```

which give an answer of

$$\frac{-\frac{(-1)^{N+2}}{2} - \frac{1}{2}}{N^2}$$

12.8 Laplace transforms

The `laplace_transform()` function in SymPy resembles the `laplace()` function in MATLAB's Symbolic Toolbox. Here we'll use it to compute the transform of

$$f(t) = e^{-at} \cos \omega t$$

Python:

```
#!/usr/bin/env python
from sympy import exp, cos, laplace_transform, symbols
from sympy.printing.latex import print_latex
```

```
s = symbols('s')
t = symbols('t')
a = symbols('a', real=True, positive=True)
omega = symbols('omega', real=True)
f = exp(-a*t)*cos(omega*t)
L = laplace_transform(f, t, s)
print_latex(L[0])
```

MATLAB:

```
sympy = py.importlib.import_module('sympy');
printing = py.importlib.import_module('sympy.printing.latex');
s = sympy.symbols('s');
t = sympy.symbols('t');
a = sympy.symbols('a', pyargs('real','True', 'positive','True'));
omega = sympy.symbols('omega', pyargs('real','True'));
f = sympy.exp(-a*t)*sympy.cos(omega*t);
L = sympy.laplace_transform(f, t, s); % a tuple, solution in 1st position
printing.print_latex(L{1})
```

which produces LaTeX that renders as

$$\frac{a+s}{\omega^2+(a+s)^2}$$

Chapter 13

Accelerate MATLAB with Python and Numba

2022/10/28

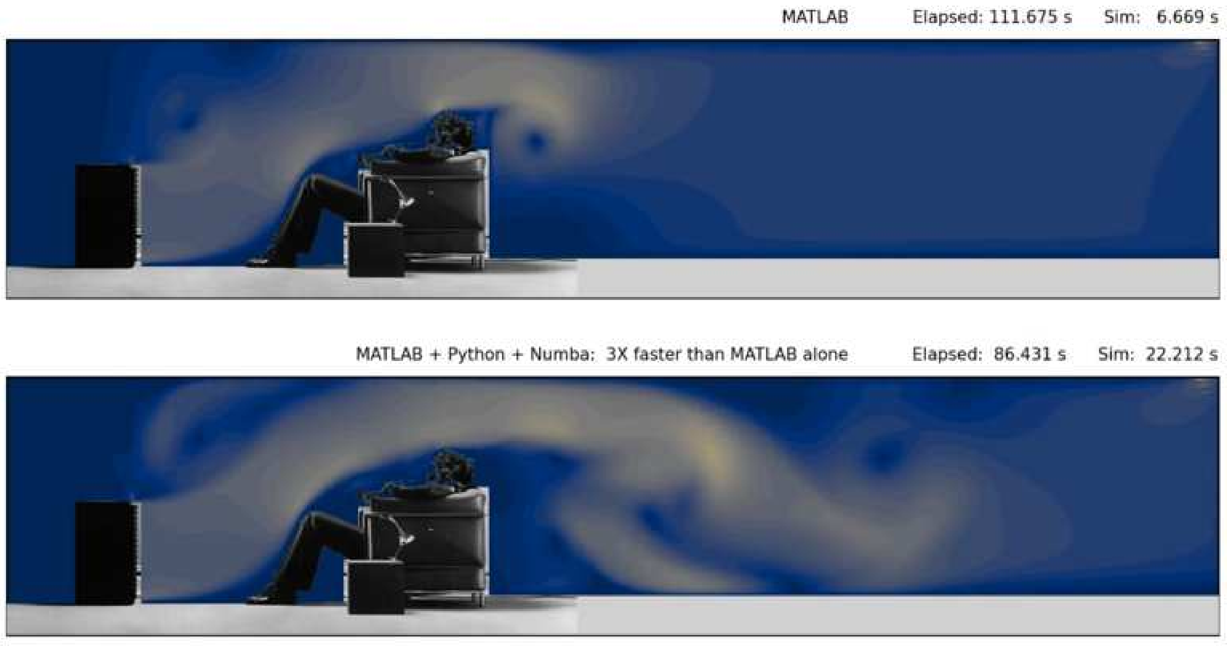


Figure 13.1: **Upper panel**: air flow computed with MATLAB 2022b using a Navier-Stokes solver by Jamie Johns. **Lower panel**: air flow computed with MATLAB 2022b calling the same code translated to Python with Numba. It repeats the simulation more than three times in the amount of time it takes the MATLAB-only solver to do this once. Animation on YouTube

13.1 Introduction

MATLAB's number crunching power is impressive. If you're working with a compute-bound application though, you probably want it to run even faster. After you've exhausted the obvious speed enhancements like using more efficient algorithms, vectorizing all operations, and running on faster hardware, your last hope may be to to rewrite the slow parts in a compiled language linked to MATLAB through mex. This is no easy task though. Re-implementing a MATLAB function in C++, Fortran, or Java is tedious at best, and completley impractical at worst if the code to be rewritten calls complex MATLAB or Toolbox functions.

In this chapter I show how Python and Numba can greatly improve MATLAB performance with less effort, and with greater flexibility, than using mex with C++, Fortran, or Java. You'll need to include Python in your MATLAB workflow (Chapter 3), but that's no more difficult than installing compilers and configuring mex build environments.

Two examples demonstrate the performance boost Python and Numba can bring MATLAB: a simple Mandelbrot set computation, and a much more involved Navier-Stokes solver for incompressible fluid flow. In both cases, the Python augmentation makes the MATLAB application run several times faster.

13.2 Python+Numba as an alternative to mex

The mex compiler front-end enables the creation of functions written in C, C++, Fortran, Java, and .Net that can be called directly from MATLAB. To use it effectively, you'll need to be proficient in one of these languages and have a supported compiler. With those in place, the fastest way to begin writing a mex-compiled function is to start with existing working code such as any of the MathWorks' mex examples and modifying it to your needs.

A big challenge to translating MATLAB code is that one line of MATLAB can translate to dozens of lines in a compiled language. Standard MATLAB statements such as

MATLAB:

```
y = max((A\b)*v',[],'all');
```

become a logistical headache in the compiled languages. Do you have access to an optimized linear algebra library? Do you know how to set up the calling arguments to the various solvers? What if the inputs are complex? What if `A` is rectangular? It is not straightforward.

Python, on the otherhand, can often match MATLAB expressions one-to-one (although the Python expressions are usually longer). The Python version of the line above is

Python:

```
y = np.max((np.linalg.solve(A,b).dot(v.T))
```

where `np.` is the prefix for the NumPy module.

If you're familiar with Python and its ecosystem of numeric and scientific modules, you'll be able to translate a MATLAB function to Python *much* faster than you can translate MATLAB to any compiled language.

For the most part, numerically intensive codes run at comparable speeds in MATLAB and Python. Simply translating MATLAB to Python will rarely give you a worthwhile performance boost. To see real acceleration you'll need to additionally use a Python code compiler such as Cython, Pythran, f2py, or Numba.

Numba offers the most bang for the buck—the greatest performance for the least amount of effort—and is the focus of this post.

13.3 What is Numba?

From the Numba project's Overview:

> Numba is a compiler for Python array and numerical functions that gives you the power to speed up your applications with high performance functions written directly in Python.
>
> Numba generates optimized machine code from pure Python code using the LLVM compiler infrastructure. With a few simple annotations, array-oriented and math-heavy Python code can be just-in-time optimized to performance similar as C, C++ and Fortran, without having to switch languages or Python interpreters.

13.4 Type casting Python function arguments

Three steps are needed to turn a conventional Python function into a much faster Numba-compiled function:

1. Import the `jit` function and all data types you plan to use from the Numba module. Also import the `prange` function if you plan to run for loops in parallel.
2. Precede your function definition with `@jit()`
3. Pass to `@jit()` type declarations for of each input argument and return value

The steps are illustrated below with a simple function that accepts a 2D array of 32 bit floating point numbers and a 64 bit floating point tolerance, then returns an unsigned 64 bit integer containing the count of terms in the array that are less than the tolerance. We'll ignore the fact that this is a simple operation in both MATLAB (`sum(A < tol)`) and Python (`np.sum(A < tol)`).)

First the plain Python version:

Python:

```python
def n_below(A, tol):
    count = 0
    nR, nC = A.shape
    for r in range(nR):
        for c in range(nC):
            if A[r,c] < tol:
                count += 1
    return count
```

Now with Numba:

Python:

```python
from numba import jit, uint64, float32, float64
@jit(uint64(float32[:,:], float64), nopython=True)
def n_below_numba(A, tol):
    count = 0
    nR, nC = A.shape
    for r in range(nR):
        for c in range(nC):
            if A[r,c] < tol:
                count += 1
    return count
```

Only the two highlighted lines were added; the body of the function did not change. Let's see what the two extra lines buy us:

```
In : A = np.random.rand(2000,3000).astype(np.float32)
In : tol = 0.5
In : %timeit n_below(A, tol)          # conventional Python
11.3 s +/- 72.3 ms per loop (mean +/- std. dev. of 7 runs, 1 loop each)

In : %timeit n_below_numba(A, tol)    # Python+Numba
9.92 ms +/- 251 µs per loop (mean +/- std. dev. of 7 runs, 100 loops each)
```

Time dropped from 11.3 seconds to 9.92 milliseconds, a factor of more than 1,000. While this probably says more about how slow Python 3.8 is for this problem [1], the Numba version is impressive. It is in fact even a bit faster than the native NumPy version:

```
In : %timeit np.sum(A < tol)
10.2 ms +/- 193 µs per loop (mean +/- std. dev. of 7 runs, 100 loops each)
```

For completeness, MATLAB 2022b does this about as quickly as NumPy:

```
>> A = single(rand(2000,3000));
>> tol = 0.5;
>> tic; sum(A < tol); toc
```

Repeating the last line a few times gives a peak (that is, minimum) time of 0.011850 seconds (11.85 ms), about the same as NumPy's 10.2 ms.

13.5 Options

The `@jit()` decorator has a number of options that affect how Python code is compiled. All four of the options below appear on every `@jit()` instance in the Navier-Stokes code. Here's an example from one of the functions in the solver:

```
@jit(float64[:,:](float64[:,:], float64),
     nopython=True, fastmath=True, parallel=True, cache=True)
def DX2(a,dn):
    # finite difference for Laplace of operator in two dimensions
    # (second deriv x + second deriv ystaggered grid)
    return (a[:-2,1:-1] + a[2:,1:-1] + a[1:-1,2:] + a[1:-1,:-2] -
           4*a[1:-1,1:-1])/(dn**2)
```

13.5.1 `nopython=True`

Numba compiled functions can work either in `object` mode, where the code can interact with the Python interpreter, or in `nopython` mode where the Python interpeter is not used. `nopython` mode excludes the Python interpreter and results in higher performance.

[1]Python has an impressive performance road-map ahead

13.5.2 `fastmath=True`

This option matches the `-Ofast` optimization switch used by compilers in the Gnu Compiler Collection. It allows the compiler to resequence floating point computations in a non IEEE 754 compliant manner—enabling significant speed increases at the risk of generating incorrect (!) results.

To use this option with confidence, compare solutions produced with and without it. Obviously if the results differ appreciably you shouldn't use this option.

13.5.3 `parallel=True`

This option tells Numba to attempt automatic parallelization of code blocks in jit compiled functions. Parallelization may be possible even if these functions do not explicitly call the `prange()` to make a `for` loop run in parallel.

Let's try this out on the simple `n_below()` function shown above:

Python:

```python
from numba import jit, prange, uint64, float32, float64
@jit(uint64(float32[:,:], float64), nopython=True, parallel=True)
def n_below_parallel(A, tol):
    count = 0
    nR, nC = A.shape
    for r in prange(nR):     # <-  parallel for loop
        for c in range(nC):
            if A[r,c] < tol:
                count += 1
    return count
```

By parallelizing the rows of `A` over cores I get twice the performance on my 4 core laptop:

```
In : %timeit n_below_parallel(A, tol)
5.96 ms ± 299 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Even more impressively, the Navier-Stokes solver runs nearly 2x more quickly with `parallel=True` on my 4 core laptop even though it has no parallel `for` loops.

13.5.4 `cache=True`

A program that includes Numba-enhanced Python functions start more slowly because the jit compiler needs to compile the functions before the program can begin running. The delay is barely noticable with small functions.

That's not the case for the more complicated jit-compiled functions in the Navier-Stokes solver. These impose a 10 second start-up delay on my laptop. The `cache=True` option tells Numba to cache the result of its compilations for immediate on subsequent runs. The code is only recompiled if any of the `@jit` decorated functions change.

13.6 Hardware, OS details

The examples were run on a 2015 Dell XPS13 laptop with 4 cores of i5-5200U CPU @ 2.2 GHz and 8 GB memory. The OS is Ubuntu 20.04. MATLAB 2022b was used but the code will run with any MATLAB version from 2020b onward.

Anaconda Python 2020.07 with Python 3.8.8 was used to permit running on older versions of MATLAB, specifically 2020b. (Yes, I'm aware Python 3.11 runs more quickly than 3.8. MATLAB does not yet support that version though.)

13.7 Example 1: Mandelbrot set

This example comes from the High Performance Computing chapter of my book. There I implement eight versions of code that compute terms of the Mandelbrot set: MATLAB, Python, Python with multiprocessing, Python with Cython, Python with Pythran, Python with f2py, Python with Numba, and finally MATLAB calling the Python/Numba functions.

The baseline MATLAB version [2] is much faster than the baseline Python implementation. However, employing any of the compiler-augmented tools like Cython, Pythran, f2py, or Numba turns the tables and makes the Python solutions much faster than MATLAB.

baseline MATLAB[2] :

```
% file: MB_main.m
main()

function [i]=nIter(c, imax)
  z = complex(0,0);
```

[2]Mike Croucher at the MathWorks provided code tweaks to improve the performance of this implementation.

```
  for i = 0:imax-1
    z = z^2 + c;
    if (real(z)^2 + imag(z)^2) > 4
        break
    end
  end
end

function [img]=MB(Re,Im,imax)
  nR = size(Im,2);
  nC = size(Re,2);
  img = zeros(nR, nC, 'uint8');
% parfor i = 1:nR % gives worse performance
  for i = 1:nR
    for j = 1:nC
      c = complex(Re(j),Im(i));
      img(i,j) = nIter(c,imax);
    end
  end
end

function [] = main()
  imax = 255;
  for N = [500 1000 2000 5000]
    tic
    nR = N; nC = N;
    Re = linspace(-0.7440, -0.7433, nC);
    Im = linspace( 0.1315,  0.1322, nR);
    img = MB(Re, Im, imax);
    fprintf('%5d %.3f\n',N,toc);
  end
end
```

baseline Python:

```
#!/usr/bin/env python3
# file: MB.py
import numpy as np
import time
def nIter(c, imax):
  z = complex(0, 0)
  for i in range(imax):
    z = z*z + c
    if z.real*z.real + z.imag*z.imag > 4:
      break
  return np.uint8(i)

def MB(Re, Im, imax):
  nR = len(Im)
  nC = len(Re)
  img = np.zeros((nR, nC), dtype=np.uint8)
  for i in range(nR):
    for j in range(nC):
      c = complex(Re[j], Im[i])
      img[i,j] = nIter(c,imax)
  return img

def main():
  imax = 255
  for N in [500,1000,2000,5000]:
    T_s = time.time()
    nR, nC = N, N
    Re = np.linspace(-0.7440, -0.7433, nC)
    Im = np.linspace( 0.1315,  0.1322, nR)
    img = MB(Re, Im, imax)
    print(N, time.time() - T_s)

if __name__ == '__main__': main()
```

The Python with Numba version is the same as the baseline Python version with five additional Numba-specific lines. An explanation of each highlighted line appears below the code:

Python with Numba:

```
#!/usr/bin/env python3
# file: MB_numba.py
import numpy as np
import time
from numba import jit, prange, uint8, int64, float64, complex128
@jit(uint8(complex128,int64), nopython=True, fastmath=True)
def nIter(c, imax):
  z = complex(0, 0)
  for i in range(imax):
    z = z*z + c
    if z.real*z.real + z.imag*z.imag > 4:
      break
  return i

@jit(uint8[:,:](float64[:], float64[:],int64), nopython=True,
16                  fastmath=True, parallel=True)
17 def MB(Re, Im, imax):
18   nR = len(Im)
19   nC = len(Re)
20   img = np.zeros((nR, nC), dtype=np.uint8)
21   for i in prange(nR):
22     for j in range(nC):
23       c = complex(Re[j], Im[i])
24       img[i,j] = nIter(c,imax)
25   return img
26
27 def main():
28   imax = 255
29   for N in [500, 1000, 2000, 5000]:
30     T_s = time.time()
31     nR, nC = N, N
32     Re = np.linspace(-0.7440, -0.7433, nC)
33     Im = np.linspace( 0.1315,  0.1322, nR)
34     img = MB(Re, Im, imax)
35     print(N, time.time() - T_s)
36
37 if __name__ == '__main__': main()
```

Line 5 imports the necessary items from the `numba` module: `jit()` is a decorator that precedes functions we want Numba to compile. `prange()` is a parallel version of the Python `range()` function. It turns regular `for` loops into parallel `for` loops where work is automatically distributed over the cores of your computer. The remaining items from `uint8` to `complex128` define data types that will be employed in function signatures.

Line 6 defines the signature of the `nIter()` function as taking a complex scalar and 64 bit integer as inputs and returning an unsigned 8 bit integer. (The remaining keyword arguments like `nopython=` were explained above.)

Lines 15 and 16 define the signature of the `MB()` function as taking a pair of 1D double precision arrays and a scalar 64 integer and returning a 2D array of unsigned 8 bit integers.

Line 21 implements a parallel `for` loop; iterations for different values of `i` are distributed over the cores.

What have we gained with the five extra Numba lines? This table shows how our three codes perform for different image sizes; times are in seconds.

N	MATLAB 2022b	Python 3.8.8	Python 3.8.8 + Numba
500	0.70	8.22	0.04
1000	2.29	32.70	0.14
2000	9.04	127.84	0.61

5000	56.77	795.41	3.51

The Python+Numba performance borders on the unbelievable—but don't take my word for it, run the three implementations yourself!

One "cheat" the Python+Numba solution enjoys is that its parallel for loop takes advantage of the four cores on my laptop. I tried MATLAB's `parfor` parallel for loop construct but, inexplicably, it runs slower than a conventional sequential `for` loop. Evidently multiple cores are only used if you have a license for the Parallel Computing Toolbox (which I don't).

13.8 MATLAB + Python + Numba

If Python can run faster with Numba, MATLAB can too. All we need to do is import the `MB_numba` module shown in the previous section and call its jit-compiled `MB()` function from MATLAB:

MATLAB:

```
% file: MB_python_numba.m
np = py.importlib.import_module('numpy');
MB_numba = py.importlib.import_module('MB_numba');
imax = int64(255);
for N = [500 1000 2000 5000]
    tic
    nR = N; nC = N;
    Re = np.array(linspace(-0.7440, -0.7433, nC));
    Im = np.array(linspace( 0.1315,  0.1322, nR));
    img = py2mat(MB_numba.MB(Re, Im, imax));
    fprintf('%5d %.3f\n',N,toc);
end
```

`MB()` from `MB_numba.py` is a Python function so it returns a Python result. To make the benchmark against the baseline MATLAB version fair, the program includes conversion of the NumPy `img` array to a MATLAB matrix (using py2mat.m) in the elapsed time. The table below repeats the MATLAB baseline times from the previous table. Numeric values in the middle and right column are elapsed time in seconds.

N	MATLAB 2022b	MATLAB 2022b + Python 3.8.8 + Numba
500	0.70	0.18
1000	2.29	0.17
2000	9.04	0.64
5000	56.77	3.85

13.9 Visualizing the result

Call MATLAB's `imshow()` or `imagesc()` functions on the `img` matrix if you want to see what the matrix looks like. For example,

MATLAB:

```
% file: MB_view.m
np = py.importlib.import_module('numpy');
MB_numba = py.importlib.import_module('MB_numba');
imax = int64(255);
nR = 2000; nC = 2000;
Re = np.array(linspace(-0.7440, -0.7433, nC));
Im = np.array(linspace( 0.1315,  0.1322, nR));
img = py2mat(MB_numba.MB(Re, Im, imax));
imagesc(img)
```

Similarly, in Python:

```
#!/usr/bin/env python3
# file: MB_view.py
import numpy as np
from MB_numba import MB
import matplotlib.pyplot as plt
imax = 255
nR, nC = 2000, 2000
Re = np.linspace(-0.7440, -0.7433, nC)
Im = np.linspace( 0.1315,  0.1322, nR)
img = MB(Re, Im, imax)
plt.imshow(img)
plt.show()
```

13.10 Example 2: incompressible fluid flow

The Mandelbrot example is useful for demonstrating how to use Numba and for giving insight to the performance boosts it can give. But useful as a computational pattern resembling real work? Not so much.

The second example represents computations as might appear in a realistic scientific application. It demonstrates MATLAB+Python+Numba with a two dimensional Navier-Stokes solver for

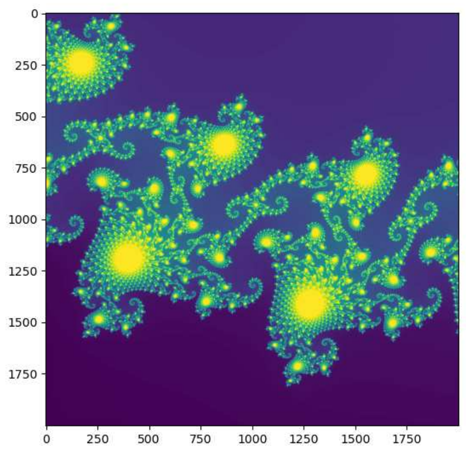


Figure 13.2: Mandelbrot set image made with matplotlib.

incompressible fluid flow. I chose an implementation by Jamie Johns which shows MATLAB at its peak performance using fully vectorized code. While this code only computes flow in 2D, the sequence of calculations and memory access patterns are representative of scientific and engineering applications. Jamie Johns' YouTube channel has interesting videos produced with his code.

13.11 Boundary conditions

A clever aspect of Jamie Johns' code is that boundary conditions are defined graphically. You create an image, for example a PNG, where colors define properties at each grid point: black = boundary, red= outflow (source), and green = inflow (sink). The image borders, if they aren't already black, also define boundaries.

This PNG image, which can be found in the original code's Github repository as `scenario_sphere2.png`, represents flow moving from left to right around a circle, a canonical fluid dynamics problem I studied as an undergrad:

Of course, if boundary conditions are supplied as images, they can be arbitrarily complex. I recalled an amusing commercial by Maxell, "Blown Away Guy", from my youth. With a bit of photo editing I converted

Figure 13.3:

Figure 13.4:

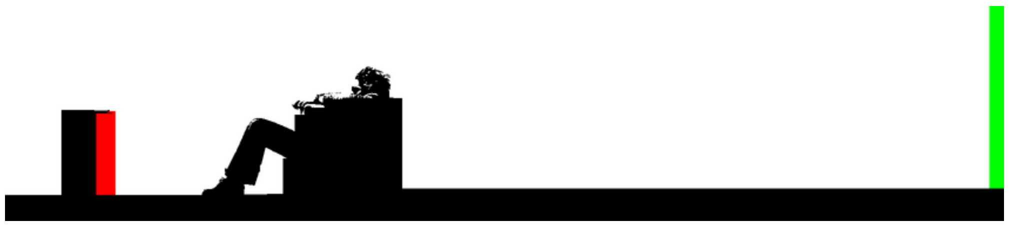


Figure 13.5:

to the following boundary conditions image:

The domain dimensions are 15 meters × 3.22 meters and the CFD mesh has 400 × 86 grid points, giving a relatively coarse resolution of 3.7 cm between points. My memory-limited (8 GB) laptop can't handle a finer mesh.

13.12 Initial conditions

- time step: 0.003 seconds
- number of iterations: 12,900
- air density = 1 kg/m^3
- dynamic viscosity = 0.001 kg/(m s)
- air speed leaving the speaker: 0.45 m/s

The low air speed is admittedly contrived and certainly lower than the commercial's video suggests. (my guess is for that is at least 2 m/s). I found the solver fails to converge, then exits with an error, if the flow speed is too high. Same thing for number of iterations—too many and the solver diverges. Trial and error brought me to 0.45 m/s and 12,900 iterations as values that produce the longest animation.

I don't understand the reason of the failure (mesh too coarse?) and haven't been motivated to figure out what's going on. This is, after all, an exercise in performance tweaking rather than studying turbulence or computing coefficients of drag.

13.13 Running the code

My modifications to Jamie Johns' MATLAB code and the translation to Python + Numba can be found at `https://github.com/AlDanial/matlab_with_python/`, under `performance/fluid_flow` (MIT license). The code there can be run three ways: entirely in MATLAB, entirely in Python, or with MATLAB calling Python.

One thing to note is that the computational codes are purely batch programs that write `.mat` and `.npy` data files. No graphics appear. The flow can be visualized with a separate post-processing step (section 13.9 below) after the flow speeds are saved as files.

13.13.1 MATLAB (2020b through 2022b)

Start MATLAB and enter the name of the main program, `Main`, at the prompt. I had to start MATLAB with `-nojvm` to suppress the GUI (and close all other applications, especially web browsers) to give the program enough memory.

```
>> Main
```

13.13.2 Python + Numba

On Linux and macOS, just type `./py_Main.py` in a console.

On Windows, open a Python-configured terminal (such as the Anaconda console) and enter `python py_Main.py`.

13.13.3 MATLAB + Python + Numba

Edit `Main.m` and change the value of `Run_Python_Solver` to `true`. Save it, then run `Main` and the MATLAB prompt.

13.13.4 Operation

Regardless of how you run the code (MATLAB / Python / MATLAB + Python), the steps of operation are the same: the application creates a subdirectory called `NS_velocity` then populates it with `.npy` files (Python and MATLAB+Python) or `.mat` files (pure MATLAB). These files contain the × and y components of the flow speed at each grid point for a batch of iterations.

The first time the Python or MATLAB+Python program starts, expect to see delay of about 30 seconds while Numba compiles the `@jit` decorated functions in `navier_stokes_functions.py`. This is a one-time cost though; subsequent runs will start much more quickly.

13.13.5 Performance

MATLAB runs more than three times faster using the Python+Numba Navier-Stokes solver. More surprisingly, Python without Numba also runs faster than MATLAB. I updated the performance table on 2022-10-30 following a LinkedIn post by Artem Lenksy asking how the recently released 3.11 version fares on this problem. Numba is not yet available for 3.11 so I reran without `@jit` decorators—and threw in a 3.8.8 run without Numba as well:

Solver	Elapsed seconds
MATLAB 2022b	458.35
Python 3.11.0	238.95
Python 3.8.8	237.46
MATLAB 2022b + Python 3.8.8 + Numba	128.89
Python 3.8.8 + Numba	126.99

Two surprises here: plain Python + NumPy is faster than MATLAB, and Python 3.11.0 is no faster than 3.8.8 for this compute-bound problem.

13.14 Visualization

The program `animate_navier_stokes.py` generates an animation of flow computed earlier with `py_Main.py` or `Main.m`. The most direct way to use it is by giving it the directory (`NS_velocity` by default) containing flow velocity files and whether you want to see `*.mat` files generated by MATLAB (use `--matlab`) or `*.npy` files generated by either Python or the MATLAB+Python combination (use `--python`). (Note: the program loads many Python modules and takes a long time to start the first time.)

```
./animate_navier_stokes.py --python NS_velocity
```

With this command the program first loads all `*.npy` files in the `NS_velocity` directory, then writes a large merged file (`merged_frames.npy`, 1.7 GB) containing the entire solution in one file. You can later load the merged file much more quickly than reading the individual `*.npy` files.

The first thing you'll notice is the animation proceeds slowly. To speed things up, skip a few frames. The next command reads from the merged frame file and only shows every 50th frame:

```
./animate_navier_stokes.py --npy merged_frames.npy --incr 50
```

Use the `--png-dir` switch to write individual frames to PNG files in the given directory. For example

```
./animate_navier_stokes.py --npy merged_frames.npy --png-dir PNG --incr 400
```

will create the directory `PNG` then populate it with 33 files with names like `PNG/frame_08000.png`. I created the animation shown at the start of this chapter by saving individual frames, overlaying the Maxell image on each, then generating an MP4 file with ffmpeg.

Chapter 14

Distribute MATLAB Computations To Python Running on a Cluster with Dask

2022/11/12

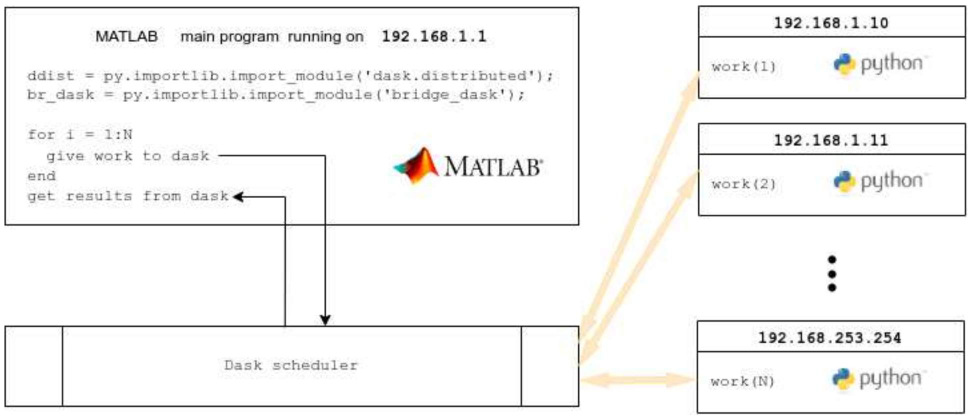
14.1 Introduction

The previous chapter showed how compute-bound MATLAB applications can be made to run faster if slow MATLAB operations are rewritten in Python. If the resulting speed improvement isn't enough, different hardware architectures like GPUs or a cluster of computers may be your only recourse for even higher performance. MATLAB has both of these nailed—if you have the Parallel Computing Toolbox and the necessary hardware.

If you don't have access to this toolbox but you do have `ssh` access to a collection of computers and you're willing to include Python in your MATLAB workflow (Chapter 3), the `dask` module can help your MATLAB application tap into those computers' collective CPU power.

14.2 Prerequisites

Although dask can run on the cores of a single computer, in this article I'll cover dask's ability to send computations to, and collect results from a group of remote computers. This raises a few complications: to use dask with a remote computer you'll need to have an account on it and be able to send commands to it without entering a password. Further, the remote computer should have enough network, file storage, memory, *and* CPU resources necessary to run your computations. In



```
ddist = py.importlib.import_module('dask.distributed');
br_dask = py.importlib.import_module('bridge_dask');

for i = 1:N
  give work to dask
end
get results from dask
```

Figure 14.1: MATLAB can run Python functions on remote computers.

particular, do you need to share the remote computer with others?

Many organizations with sizeable engineering or research departments operate compute clusters explicitly to enable large scale parallel computations. Access and resource sharing are done with job schedulers such as LSF, PBS, or Slurm. Dask has interfaces to these but here I'll work with the most simple setups of all, a group of individual computers on which I have accounts and can access through `ssh`. Here's the full list of prerequisites:

- compatible (preferrably identical) Python installations including `dask`, `distributed`, and `paramiko` modules
- the same path to the Python executable
- the same account name for the user setting up and using the cluster
- uniform bi-directional, key-based `ssh` access across all computers for the user account
- same `ssh` port
- firewall rules open to allow `ssh` and the dask scheduler and metrics ports (8786 and 8787 by default)

14.3 Start the dask cluster

Each computer that is to participate in your computations needs to have a dask *worker* process running on it. Additionally, one of the computers must run the dask *scheduler* process. These processes can be started with `ssh`, Kubernetes, Helm, or any of the job schedulers (LSF, etc) mentioned above. Any method other than `ssh`, however, will likely require help from a cluster administrator or tech support expert.

I'll avoid complications and only describe the `ssh` method.

> ***Note:*** the method described below of starting a dask cluster with `dask-ssh --hostfile` is insecure because anyone with access to the computers and knowledge of the dask port can submit jobs as though they were you. The secure way to set up a dask cluster is with a dask gateway.

Start by creating a text file with each host's IP address (or hostname, if name resolution works) on a single line. The file might look like this

```
# file: my_hosts.txt
127.0.0.1
192.168.1.39
192.168.1.40
192.168.1.41
192.168.1.65
```

Hostnames may be repeated. I typically run with a file that defines only localhost entries such as

```
# file: my_hosts.txt
127.0.0.1
127.0.0.1
127.0.0.1
```

while I'm developing a parallel program. I only submit to remote computers after I'm convinced the code works locally.

Next, pass the file of host names to the `dask-ssh` command to start the worker on each node, and also a scheduler:

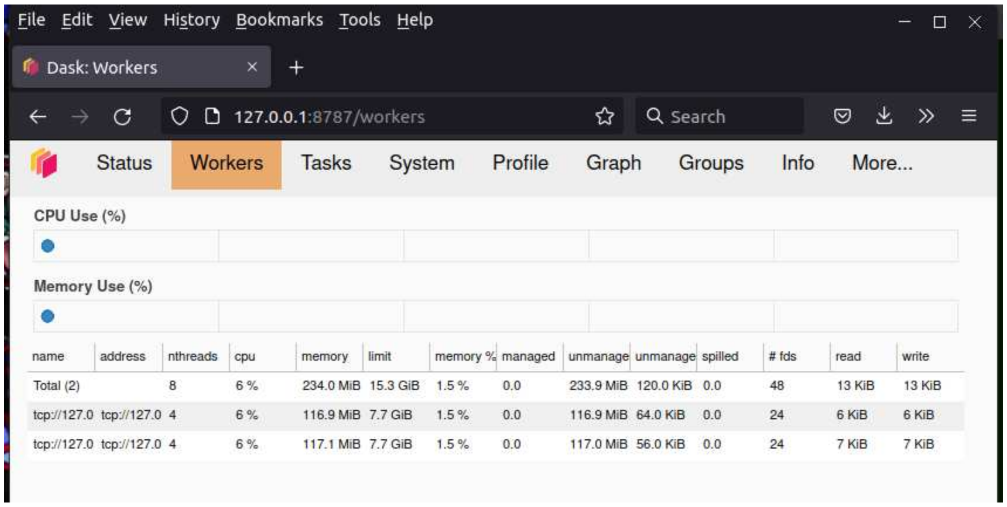
```
dask-ssh --hostfile my_hosts.txt
```

You'll see a stream of messages resembling

```
---------------------------------------------------------------
                 Dask.distributed v2022.7.0

Worker nodes: 3
  0: 127.0.0.1
  1: 127.0.0.1
  2: 127.0.0.1

scheduler node: 127.0.0.1:8786
```



name	address	nthreads	cpu	memory	limit	memory %	managed	unmanage	unmanage	spilled	# fds	read	write
Total (2)		8	6 %	234.0 MiB	15.3 GiB	1.5 %	0.0	233.9 MiB	120.0 KiB	0.0	48	13 KiB	13 KiB
tcp://127.0	tcp://127.0	4	6 %	116.9 MiB	7.7 GiB	1.5 %	0.0	116.9 MiB	64.0 KiB	0.0	24	6 KiB	6 KiB
tcp://127.0	tcp://127.0	4	6 %	117.1 MiB	7.7 GiB	1.5 %	0.0	117.0 MiB	56.0 KiB	0.0	24	7 KiB	7 KiB

```
---------------------------------------------------------------


/usr/local/anaconda3/2021.05/lib/python3.8/site-packages/paramiko/transport.py:219: CryptographyDeprecationWarning:
  "class": algorithms.Blowfish,
[ scheduler 127.0.0.1:8786 ] : /usr/local/anaconda3/2021.05/bin/python -m distributed.cli.dask_scheduler --port 878
                                               :
                                (lines deleted)
                                               :
INFO - Register worker <WorkerState 'tcp://127.0.0.1:37661', status: init, memory: 0, processing: 0>
[ scheduler 127.0.0.1:8786 ] : 2022-11-07 20:58:36,128 - distributed.scheduler - INFO - Starting worker compute str
[ scheduler 127.0.0.1:8786 ] : 2022-11-07 20:58:36,129 - distributed.core - INFO - Starting established connection
```

Note line of output with `scheduler node: 127.0.0.1:8786`. This is the host and port where the scheduler listens for instructions. We'll need this value to create a `dask` client (described below).

The `dask-ssh` command will tie up your terminal until the cluster is shut down. That can be done by entering *ctrl*-c in the terminal.

The dask scheduler runs a web server that shows lots of useful information about the state of your cluster. View it by entering your scheduler node's hostname or IP address followed by `:8787` as a URL in your browser. In my case that's `http://127.0.0.1:8787` or `http://localhost:8787`. The view under the Workers tab looks like this:

Track your compute nodes on this web while dask is cranking away at one of the following examples, all of which come from the High Performance Computing chapter of my book.

14.4 Example 1: sum of prime factors

I'll start with a simple example to demonstrate the mechanics of using dask. The Python and MATLAB programs below compute the sum of unique prime factors of all numbers in a given range.

The hard part, computing the prime factors themselves, is done with SymPy's `primefactors()` function, comparable to MATLAB's `factor()`[1].

Here are sequential implementations:

Python:

```python
#!/usr/bin/env python3
# file: prime_seq.py
from sympy import primefactors
import time
def my_fn(a, b, incr):
    Ts = time.time()
    s = 0
    for x in range(a,b,incr):
        s += sum(primefactors(x))
    return s, time.time() - Ts
def main():
    A = 2
    B = 10_000_000
    incr = 1
    S, dT = my_fn(A, B, incr)
    print(f'A={A} B={B} {dT:.4f} sec')
    print(f'S={S}')
if __name__ == "__main__": main()
```

MATLAB:

```matlab
% file: prime_seq.m
A = 2;
B = 10000000;
incr = 1;
[S, dT] = my_fn(A, B, incr);
fprintf('A=%d B=%d, %.3f sec\n', A, B, dT);
fprintf('S=%ld\n', S);
function [S, dT]=my_fn(A, B, incr)
    tic;
    S = 0;
    for i = A:incr:B-1
        S = S + sum(unique(factor(i)));
    end
    dT = toc;
end
```

Performance looks like this on my laptop:

Language	Elapsed seconds	Computed sum
MATLAB 2022b	546.737	5495501355056
Python 3.8.8	519.531	5495501355056

Calls to the compute intensive function `my_fn()` are independent and can be called in any order, or even simultaneously. We'll do exactly that with dask, using three cores of the local machine. To spread the load evenly we merely need to offset the first number in the sequence by the job number, 0 to N-1 for N jobs, then stride through the sequence in steps of N. For example if we compute prime factors for numbers between 2 and 20 and want to spread the work evenly over three jobs, the assignments will be

```
In : list(range(2,21))            # original set
Out: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

In : list(range(2+0,21,3))        # for processor 1 (does job 0)
Out: [2, 5, 8, 11, 14, 17, 20]

In : list(range(2+1,21,3))        # for processor 2 (does job 1)
Out: [3, 6, 9, 12, 15, 18]

In : list(range(2+2,21,3))        # for processor 3 (does job 2)
Out: [4, 7, 10, 13, 16, 19]
```

or, equivalently in MATLAB,

```
>> 2+0:3:20                       % for processor 1 (does job 0)
     2     5     8    11    14    17    20

>> 2+1:3:20                       % for processor 2 (does job 1)
     3     6     9    12    15    18

>> 2+2:3:20                       % for processor 3 (does job 2)
     4     7    10    13    16    19
```

The dask-enabled version of the prime factor summation program on the local computer, splitting up the work to three tasks that are to run simultaneously, looks like this:

```
#!/usr/bin/env python3
# file: prime_dask.py
from sympy import primefactors
import time
```

```
 5 import dask
 6 from dask.distributed import Client
 7 
 8 def my_fn(a, b, incr):
 9     Ts = time.time()
10     s = 0
11     for x in range(a,b,incr):
12         s += sum(primefactors(x))
13     return s, time.time() - Ts
14 
15 def main():
16     client = Client('127.0.0.1:8786')
17     tasks  = []
18     main_T_start = time.time()
19     n_jobs = 3
20     A = 2
21     B = 10_00 # 0_000
22     incr = n_jobs
23     for i in range(n_jobs):
24         job = dask.delayed(my_fn)(A+i, B, incr)
25         tasks.append(job)
26     results = dask.compute(*tasks)
27     client.close()
28     total_sum = 0
29     for i in range(len(results)):
30         partial_sum, T_el = results[i]
31         print(f'job {i}:  sum= {partial_sum}  T= {T_el:.3f}')
32         total_sum += partial_sum
33     print(f'total sum={total_sum}')
34     elapsed = time.time() - main_T_start
35     print(f'main took {elapsed:.3f} sec')
36 if __name__ == "__main__": main()
```

Lines 5, 6 import the `dask` module and the `Client` class

Line 16 Creates a `Client` object that connects to the dask scheduler on the cluster we set up in section 14.3.

Line 17 Initializes an empty list that will later store "tasks", basically a unit of work made of a function to call and its input arguments.

Line 19 defines the number of chucks we'll subdivide the work into. This number should match or exceed the number of dask workers, in other words, the number of remote computers we can run on.

Lines 23-25 populate the list `tasks`. Each entry, "`job`", is an invocation of `my_fn()` with input arguments that stride between `A` and `B` so as to span a unique collection of numbers.

Line 26 tells the scheduler to send the tasks to the workers to begin performing the computations. The return value from each task comes back as the list `results`.

Line 27 is reached when the last worker has finished.

Lines 28-32 fetches the solution for each task and adds it to the global sum variable.

With the scheduler and workers running in the background, we can now run the dask-enabled prime factor summation program `prime_dask.py`. It runs faster than the sequential version, but not by much:

```
3 worker processes, 3 dask jobs:
job 0: sum= 2418248946959 T= 327.969
job 1: sum= 659400211060 T= 206.228
job 2: sum= 2417852197037 T= 327.709
total sum=5495501355056
main took 328.852 sec
```

Performance disappoints for two reasons. First, the workload is clearly imbalanced since one job took around 200 seconds, while the other two took more than 320. Evidently, terms of the sequence 3, 6, 9 ... can be factored much more rapidly than terms in 2, 5, 8 ... and 4, 7, 10 ...—who knew? The second reason is less obvious. It is that the sum of individual core times, 328.0 + 206.2 + 327.7 = 861.9 seconds, is 66% higher than the single-core time of 519.5 seconds. Clearly there's a non-trivial overhead to using dask.

14.4.1 Parallel prime factor summation in MATLAB

Now that we can run our Python program in parallel with dask, we can do the same thing in MATLAB—with a couple of restrictions:

1. Tasks submitted to dask must be Python functions. Therefore our MATLAB main program will need to send the Python version of the compute function, `my_fn()` in this example, to the remote workers.
2. MATLAB cannot call the `dask.delayed()` and `dask.compute()` functions directly. In particular, the asterisk in the call to `dask.compute()` (see line 26 in the listing above) is not legal MATLAB syntax. We'll need a small Python bridge module to overcome these limitations.

The first restriction isn't a big deal for this example; we merely need to import `my_fn()` from `prime_seq.py` or `prime_dask.py`. For a more substantial MATLAB application, though, this could mean translating a lot of MATLAB code to Python.

14.4.2 Aside: `bridge_dask.py`

The second restriction is easily resolved with just a few lines of Python:

```python
# file: bridge_dask.py
import dask
def delayed(Fn, *args):
    return dask.delayed(Fn)(*args);
def compute(delayed_tasks):
    return dask.compute(*delayed_tasks)
```

This simple module provides wrappers to `dask.delayed()` and `dask.compute()` in a form suitable for MATLAB.

14.4.3 Parallel prime factor summation in MATLAB, continued

We now have the pieces in place to have MATLAB send the prime factor computation and summation work to a cluster of workers. The solution looks like this:

```matlab
% file: prime_dask.m
ddist = py.importlib.import_module('dask.distributed');
prime_seq = py.importlib.import_module('prime_seq');
br_dask = py.importlib.import_module('bridge_dask');

A = int64(2);
B = int64(10000000);
n_jobs = int64(3);

tic;
client = ddist.Client('127.0.0.1:8786');
client.upload_file('prime_seq.py');

% submit the jobs to the cluster
tasks = py.list({});
incr = n_jobs;
for i = 0:n_jobs-1
  task = br_dask.delayed(prime_seq.my_fn, A+i, B, incr);
  tasks.append(task);
end
results = br_dask.compute(tasks);  % start the computations,
client.close()      % worker's solution to the list 'results'
% gets here after all remote workers have finished
```

```
24
25  % post-processing:  aggregate the partial solutions
26  S = 0;
27  mat_results = py2mat(results);
28  for i = 1:length(mat_results)
29    partial_sum = mat_results{i}{1};
30    S = S + partial_sum;
31    fprintf('Job %d took %.3f sec\n', i-1, mat_results{i}{2});
32  end
33  dT = toc;
34  fprintf('A=%d B=%d, %.3f sec\n', A, B, dT);
35  fprintf('S=%ld\n', S);
```

Line 12 has an important we didn't need in the pure Python solution: it uploads our Python module file to the workers. Keep in mind that the workers do not share memory or file resources with the main program and therefore need to be given all items, whether input data or our code files, to function properly. (Incidentally, the `upload_file()` function can only be used to send source files to the workers. A different function, demonstrated in Example 3, is needed for other files.)

Our bridge module functions are called at **lines 18 and 21**. **Line 26** calls the py2mat.m function to convert the Python solution list into a MATLAB cell array.

MATLAB's parallel performance matches Python's—mediocre, not great:

```
>> prime_dask
Job 0 took 345.014 sec
Job 1 took 219.340 sec
Job 2 took 344.900 sec
A=2 B=10000000, 345.164 sec
S=5495501355056
```

Nonetheless we've established the mechanics of sending computations from a MATLAB main program to a collection of Python workers and getting the results back as native MATLAB variables.

14.5 Example 2: gigapixel Mandelbrot image

My second example does a bit more interesting work in a bit more interesting fashion: I'll scale up the Mandelbrot set example from 5,000 × 5,000 pixels to 35,000 × 35,000—more than 1.2 billion pixels. This is 49x larger than the 5k × 5k image so we can expect it to take 49x longer—several minutes instead of several seconds. This time I'll farm the work out to actual remote computers, a collection of 18 virtual machine instances from DigitalOcean[2]. Foolishly, perhaps, I start the dask cluster when I need it using the insecure `dask-ssh --hostname` method.

I had to restructure the solution to work on the memory limited workers. Instead of simply subdividing my 35,000 × 35,000 frame across the 18 workers, I had to go an extra step and first break the frame into smaller sub-frames, spread each sub-frame across the 18 workers, harvest the results from the sub-frame, then send the next sub-frame out. The code below uses the term `group` to refer to a sub-frame:

MATLAB:

```
% file: MB_dask.m
ddist = py.importlib.import_module('dask.distributed');
np    = py.importlib.import_module('numpy');
MB_py = py.importlib.import_module('MB_numba_dask');
br_dask = py.importlib.import_module('bridge_dask');

N = int64( 35000 );
imax = int64(255);
Tc = tic;
nR = N; nC = N;
Re_limits = np.array({-0.7440, -0.7433});
Im_limits = np.array({ 0.1315,  0.1322});
n_groups = 16;
n_jobs_per_group = 18;
n_jobs = n_jobs_per_group*n_groups;
client = ddist.Client('127.0.0.1:8786'); % replace with scheduler IP
client.upload_file('MB_numba_dask.py');
results = py.list({});
for c = 1:n_groups
  tasks  = py.list({});
  Tcs = tic;
  for i = 1:n_jobs_per_group
    job_id = int64((c-1)*n_jobs_per_group+i-1);
    task = br_dask.delayed(MB_py.MB, nC, Re_limits, ...
        nR, Im_limits, imax, job_id, n_jobs);
    tasks.append(task);
  end
  results.extend( br_dask.compute(tasks) );
  Tce = toc(Tcs);
  fprintf('group %2d took %.3f sec\n', c-1, Tce)
end
client.close();

% reassemble the image
```

```
35  img = zeros(N,N, 'uint8');
36  Tps = tic;
37  for i = 1:length(results)
38    my_rows  = py2mat(results{i}{1}) + 1; % 0-indexing to 1
39    img_rows = py2mat(results{i}{2});
40    img(my_rows,:) = img_rows;
41  end
42  Tpe = toc(Tps);
43  fprintf('py2mat conversion took %.3f\n', Tpe);
44  Te = toc(Tc);
45  fprintf('%.3f  %5d\n', Te, N);
46  imshow(img)
```

Line 4 imports the Numba-powered Mandelbrot functions from the file MB_numba_dask.py. The MB() function in this file differs a bit from the one in the previous chapter. This MB() figures out which rows of the sub-frame to stride over then returns the indices of these rows to the caller so that the caller can reassemble the overall frame more easily.

Line 16 shows the IP address of the localhost. In reality, I use the IP address of the DigitalOcean instance running my dask scheduler. I won't advertise this address to keep my instances from unnecessary attention.

Line 17 uploads the MB_numba_dask.py module file to the workers.

The loop starting at **line 19** iterates over the 16 sub-frames. I further divide each sub-frame across 18 workers so a worker at any one time is only doing math on 35000/(18*16), or roughly 122 rows at a time (each of which has 35000 columns).

Line 28 collects the results for the sub-frame from the 18 workers, then the loop continues to the next sub-frame.

Line 38 and 39 convert the Python numeric solution into native MATLAB variables. These are expensive steps because more than a gigabyte of data has to be copied.

Line 46 displays the resulting image, which doesn't seem noteworthy. It is, though. MATLAB on an 8 GB laptop can comfortably render the gigapixal image whereas matplotlib's equivalent command, plt.imshow() crashes.

Here are performance numbers for a typical run:

```
>> MB_numba_dask
group 0 took 5.658 sec
group 1 took 6.278 sec
group 2 took 4.812 sec
group 3 took 5.495 sec
group 4 took 4.746 sec
group 5 took 5.028 sec
group 6 took 4.982 sec
group 7 took 5.098 sec
group 8 took 4.911 sec
group 9 took 5.036 sec
group 10 took 4.923 sec
group 11 took 4.790 sec
group 12 took 4.946 sec
group 13 took 4.530 sec
group 14 took 4.528 sec
group 15 took 5.192 sec
py2mat conversion took 30.253
111.463 35000
```

In other words, just under 2 minutes to get results when running on 18 computers compared to an estimated 46 minutes for a pure MATLAB solution running on a single computer (56.77 seconds for 5k $\times$ 5k * 49 larger problem / 60 sec/min, ref. section 13.7). True, much of the performance comes just from Numba. However the DigitalOcean virtual machines have just one core each and these run much slower than the cores of my laptop. I can't run MATLAB on the virtual machines so it is difficult to separate know what my parallel efficiency is.

14.6 Example 3: finite element frequency domain response

Few real-world problems are as clean to set up and solve as the previous two examples. Realistic tasks typically involve substantial amounts of input data and then create volumes of new data. Figuring out how to split, transfer, then reassemble data efficiently can be as difficult as designing algorithms that can run in parallel. Often, as in this next example, you have to decide whether to transfer data from the main program to the remote workers or have all the workers perform duplicate work to generate the same data. The problem is compounded when using MATLAB with dask because you'll also need to decide how much of your MATLAB code you're willing to reimplement in Python for execution on the remote computers.

This example comes from the field of structural dynamics. It solves a system of equations involving stiffness (K), damping (C), and mass (M) matrices from a finite element discretization of a structure at many frequencies ω_k, where $k = 1 .. N$:

$$K x_k + i\omega_k C x_k - \omega_k^2 M x_k = P(\omega_k) b$$

N is typically in the hundreds but can exceed a thousand if the frequency content of the load spectrum, $P(\omega_k)$, spans a wide range. The vector b holds the load vector orientation.

If the finite element model is large, each individual solution will take minutes (or worse!). Scale that by N and you'll be waiting a while for answers, possibly a very long while. The problem is easily divided among processors, however, since each frequency's solution is independent of the other frequencies.

The first design decision to make is should we 1) compute K, C, and M in the main program, written in MATLAB, then copy the matrices to the remote computers for solution in Python, or 2) send the model description to the remote computers and have them independently regenerate K, C, and M, then solve them at a subset of frequencies?

The second method involves much less network traffic but it also means translating the MATLAB finite element generation and assembly routines to Python. The first method puts a big load on the network to transfer the large matrices to many remote computers but it also means there's little extra code to write; Python already has fast linear equation solvers for dense and sparse matrices. I'll choose the lazy "write less code" option, method 1, in this case.

Chapter 14 of my book describes the full finite element solver using sparse matrices implemented in both MATLAB and Python and includes representative models of a notional satellite with >600,000 degrees of freedom (DOF). The elements are merely 2D rods; I made no attempt to emulate a realistic structure. Multiple mesh discretizations were created with triangle. A 7,600 DOF version (`satellite.2.ele` and `satellite.2.node`) looks like this:

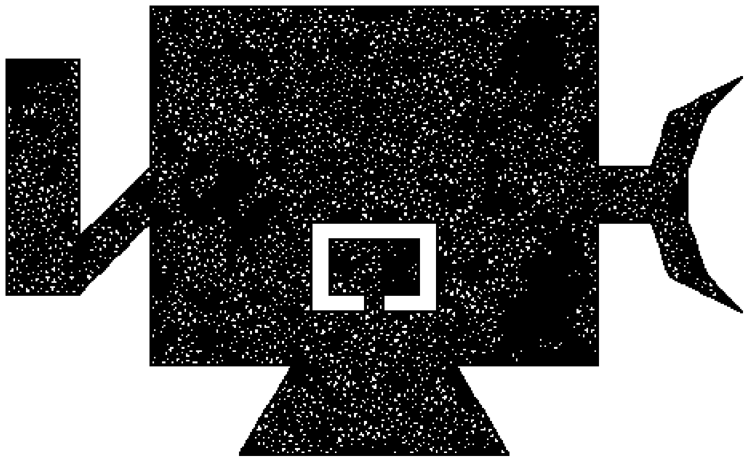


Figure 14.2:

The load represents an explosive bolt in the elbow of the solar array deployment structure on the left. Its time history and spectral content are

Rather than repeating the details here I'll describe problems I encountered trying to solve the 600k DOF model in MATLAB + dask across 2,048 frequencies using 100 remote workers of Coiled's cloud.

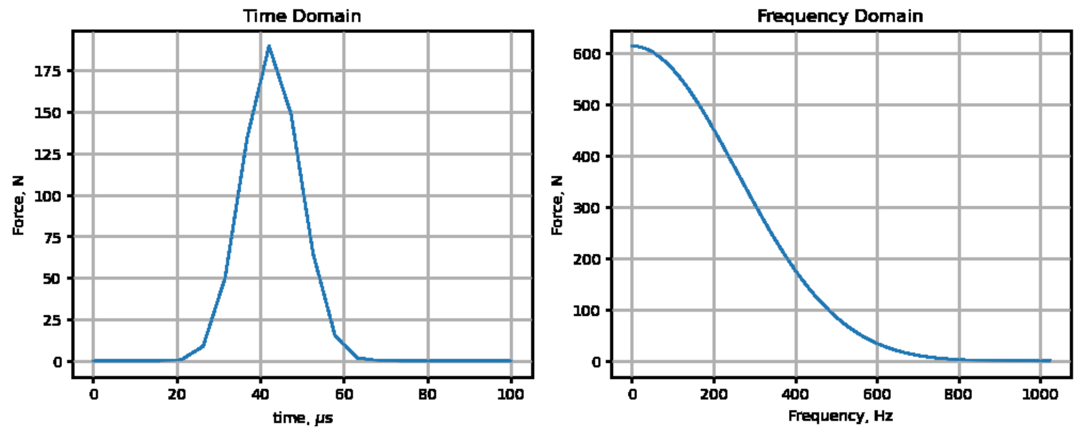


Figure 14.3:

14.6.1 Aside: Coiled

Coiled is a company created by dask's founders. It offers dask as a cloud service. Using this service is much easier path than creating a roll-your-own cluster as I did with my 18 anemic DigitalOcean virtual machines. In addition to providing computationally powerful computers, Coiled gives users 10,000 hours of free CPU each month.

I ran this example on Coiled's cloud.

14.6.2 Sending large amounts of input data to remote workers

If we have a function that takes two input arguments:

```
result = my_function(argument_1, argument_2)
```

we can send multiple copies of it to run simultaneously on remote computers with

```
tasks = []
task = client.submit(my_function, argument_1, argument_2)
tasks.append(task)
task = client.submit(my_function, argument_3, argument_4)
```

```
tasks.append(task)
            #   :
            # and so on
            #   :
results = client.gather(tasks)
```

Dask does many things behind the scenes to make this work. It

1. finds an idle worker to run the function
2. serializes the function arguments then transmits them over the network to the worker
3. deserialized the arguments then instructs the worker to call the function with these arguments
4. captures the return values from the function, serializes them, then transmits them back to the main program where it deserializes them and adds them to the `results` list

The second step is problematic when the input arguments are massive and don't change between function calls. In our case the input matrices K, C, M, P, and b are invariant and should only be sent over the network once.

Dask has a function for this situation called `scatter()` which sends the given arguments to all remote computers. It works best when these workers are ready to receive data, so we'll use another convenience function, `wait_for_workers()`. The before and after calls are arranged like this:

Python (regular way, without scatter):

```python
tasks = []
client = Client(cluster)
for i in range(n_jobs):
        task = client.submit(solve, K, C, M, b, omega_subset[i])
        tasks.append(task)
results = client.gather(tasks)
```

Python (with scatter):

```python
tasks = []
client = Client(cluster)
client.wait_for_workers(n_jobs)
KCMb_dist = client.scatter([K, C, M, b], broadcast=True)
for i in range(n_jobs):
        task = client.submit(solve, KCMb_dist, omega_subset[i])
        tasks.append(task)
results = client.gather(tasks)
```

14.6.3 MATLAB client disconnects

The next problem I encountered was disconnects between the Python worker processes on the remote computers and the dask client in the MATLAB main program. I still don't know the cause for this although it may be related to the amount of time the computations took on the remote workers.

The solution I came up with was another Python bridge module that contained all dask objects and function calls. Once all dask activity happened in Python, the MATLAB main program ran successfully.

14.6.4 Source code

The MATLAB code to compute the finite element matrices and run the direct frequency domain problem along with the Python bridge module and equation solver are too lengthy to list here. Instead you can find them at my book's Github repository in these locations:

Operation	source file name
MATLAB main program	`code/dask/run_fr_dask.m`
Python main program	`code/dask/run_fr_dask.py`
bridge module + solver	`code/dask/pysolve.py`
compute K,M	`code/mesh/FE_model.m`
compute K,M	`code/mesh/Node.m`
compute K,M	`code/mesh/Rod_Elem.m`
compute K,M	`code/mesh/load_model.m`
compute K,M	`code/mesh/run_fem.m`
satellite FE model	`satellite`

14.6.5 Performance and cost

The time to solve 2048 frequencies on my 600k DOF model on a single computer in MATLAB is about 8.5 hours. When the MATLAB program distributes the matrices and equation solving step to 100 workers on Coiled's cloud, the time drops to about 15 minutes, a 33x speed increase.

Number of Workers	Number of Frequencies	Time [seconds]	Cost [US$]
3	16	402.1	0.02
64	64	472.1	0.50
100	100	628.0	0.88
100	1024	743.0	0.95
100	2048	940.2	1.60

14.6.6 Concluding remarks

Dask does a lot of work under the hood to make it easy to send work to remote computers. It manages network transmissions of data to and from the workers and balances the load to keep all workers busy. The price for this convenience is a relatively high overhead when tasks are short, that is, less than 10 seconds each. Examples 1 and 2 fall in this relatively wimpy problem class and don't show dask at its best.

Dask shines on large problems that would otherwise overwhelm a single computer.