Open in Colab

Lab 3: Debugging and profiling code

Components

- TA demonstration
  - The TA will show his/her debugging and profiling setup in VS Code.

1. Debug code
   - Specifically, we'll have the debugger stop inside someone else's code!
2. Profiling code
   - Walk through a short demo
   - Students will then profile code themselves
   - Show a few alternative (hacky) methods

Learning objectives

- Understand that debuggers and profilers exist and make your life easier
- How to use a debugger within a Jupyter notebook
- How to profile code

Intro

As our example today, we'll look at code for computing the matrix exponential, e^A. You should remember this from taking matrix methods APPM 3310. If A is diagonalizable, then we can compute e^A using eigenvalues; if A is not diagonalizable, it's a bit more complicated. Regardless, computing e^A *efficiently* and *stably* is somewhat challenging and is a classic problem in numerical analysis.

We'll use the standard `scipy.linalg` package to do this for us, and demonstrate how we can debug and profile code, even when we didn't write the code ourselves!

This lab was tested with SciPy versions 1.11.1 and 1.16.1. We *hope* it works with most other recent versions!

Note: we recommend you download this entire `.ipynb` notebook and run it locally on your computer. You can download it via navigating our github site, or directly from `https://raw.githubusercontent.com/cu-applied-math/appm-4600-numerics/refs/heads/`

main/Labs/Lab03_DebuggingAndProfiling.ipynb

In [1]:
```
import scipy
print(scipy.__version__)
```

```
1.17.0
```

1. Debugging

Note: we **highly** suggest running this in an editor like VS Code, or at least Jupyter Lab (not a plain Jupyter Notebook). You can debug in colab, but it's actually harder because you have to use some command line debugging skills. See this article Debugging in Google Colab if you're curious (it uses the `ipdb` package).

Note: If using a local editor, like VSCode, there may be a setting that by default does **not** let you debug code from libraries/packages. To make sure you can debug the scipy code, do the following:

1. Open the VSCode settings (one way to do this is to click on the gear icon in the lower left hand corner).
2. In the settings search bar, search "debug just my code".
3. This should return a setting/checkbox tilted "Jupyter: Debug Just My Code". By default, this is checked/enabled. **Uncheck/disable** this setting.

Now you should be able to debug the scipy code.

There may be issues in Jupyter Lab with debugging external code as well. If you really cannot get it to work, let the TA know, and instead write your own code and step through it with the debugger.

Instructions

Run the following code using your IDE, like VS Code. Inside the `expm` function, there is a line of code `m, s = pick_pade_structure(Am)`.

Deliverable

For the matrix A below, when it is passed into `expm`, what is the value of `m` from the `m, s = pick_pade_structure(Am)` line inside the Scipy code?

In [3]:
```python
import scipy.linalg as sla
import numpy as np

A = np.arange(int(1e2)).reshape(10,10)/100
print(A)
eA = sla.expm(A)
```

```
[[0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09]
 [0.1  0.11 0.12 0.13 0.14 0.15 0.16 0.17 0.18 0.19]
 [0.2  0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29]
 [0.3  0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39]
 [0.4  0.41 0.42 0.43 0.44 0.45 0.46 0.47 0.48 0.49]
 [0.5  0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59]
 [0.6  0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69]
 [0.7  0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79]
 [0.8  0.81 0.82 0.83 0.84 0.85 0.86 0.87 0.88 0.89]
 [0.9  0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99]]
```

From debugging, $m = 13$ for this program

2. Profiling Code

Profiling code refers to looking at running code and noticing what parts of the code take a long time, as well as memory usage. For memory, you might look at what parts of the code require a lot of memory, or where memory leaks are, or at the cache miss rate, etc.

We're not going to focus on memory profiling, but instead on time (aka speed) profiling. A profiler tells us which parts of the code took a long time. Some profiling tells us which **functions** take a while, other types of profiling tell us which **lines** of code take a while. Today we'll do the latter.

There are some caveats: to do profiling, the profiler adds some extra overhead, so everything runs a bit slower, and you don't get a 100% accurate picture of exactly how long each part of the code takes. Usually it's accurate enough to give you an idea of where the slowish parts are.

Some programmers avoid formal profiling but just adding timing statements into their code (see section 2c for a few ways to do this). Sometimes this is easy to do and sufficient, but other times it's not systematic enough and takes more effort than you need; it's also a problem if you want to profile some library code where you don't want to have to edit the files.

2a. Example usage

We're going to use the `line_profiler` package, so let's first check if it is installed (and if not, we'll install it):

In [5]:
```python
try:
```

```
    import line_profiler
    print(f"The package 'line_profiler' is installed.")
except ModuleNotFoundError:
    print(f"The package 'line_profiler' is NOT installed.")
    # To install it via PIP, we can do the following:
    import subprocess
    subprocess.check_call(['pip', 'install', 'line_profiler'])
    # or, just run: !pip install line_profiler     from within jupyter
    # or, just run:  pip install line_profiler     from a command line
    # or, if using conda, run:   conda install conda-forge::line_profiler     from a command line
```

```
The package 'line_profiler' is installed.
```

Now let's use it. There are different ways to use it. We'll use it in one manner which works well with jupuyter notedbooks. The following is an ipython/jupyter specific command (you only need to run this once per session):

In [6]:
```
%load_ext line_profiler
```

Now let's have some code to profile. Below is some silly code that does the computation

$$f(n) = \sum_{k=1}^{n} k$$

in a few different ways.

In [22]:
```
def sum_python(n):
  sum = 0.
  for k in range(n+1):
      sum += k
  return sum

def sum_numpy(n):
  nList = np.arange(n+1)
  sum   = np.sum(nList)
  return sum

def my_sum(n):
  """ returns the sum of k from k = 1 ... n """
  s1 = sum_python(n)
  s2 = sum_numpy(n)
  s3 = n*(n+1)/2  # use the closed-form formula, cf.  https://en.wikipedia.org/wiki/1_%2B_2_%2B_3_%2B_4_%2B_%E2%8B%AF
  assert np.isclose(s1,s2), "The first two methods gave different values :-( )"
  assert np.isclose(s2,s3), "The second two methods gave different values :-( )"

  return s1
```

```
my_sum(1000)
```

Out[22]: 500500.0

Now let's see how to use the line profiler, which will tell us which lines are taking up most of the time.

Note: this works better in an IDE like VS Code. It *kinda* works in Colab (it may only tell you the line number)

We'll use it in the following form: `%lprun -f name_of_fcn  big_fcn(...)` where `big_fcn(...)` is the code that you are going to call, and `name_of_fcn` is the *part* of that code that you want to investigate line-by-line.

For example:

In [18]:
```
%lprun -f my_sum my_sum(1000)
```

```
Timer unit: 1e-09 s

Total time: 0.000917 s
File: /var/folders/40/26f5s8qj42z2mxw1gfrnh5rm0000gn/T/ipykernel_50763/4186063441.py
Function: my_sum at line 12

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    12                                           def my_sum(n):
    13                                             """ returns the sum of k from k = 1 ... n """
    14         1     657000.0 657000.0     71.6    s1 = sum_python(n)
    15         1      70000.0  70000.0      7.6    s2 = sum_numpy(n)
    16         1          0.0      0.0      0.0    s3 = n*(n+1)/2  # use the closed-form formula, cf.  https://en.wikip
edia.org/wiki/1_%2B_2_%2B_3_%2B_4_%2B_%E2%8B%AF
    17         1     122000.0 122000.0     13.3    assert np.isclose(s1,s2), "The first two methods gave different valu
es :-( )"
    18         1      67000.0  67000.0      7.3    assert np.isclose(s2,s3), "The second two methods gave different val
ues :-( )"
    19
    20         1       1000.0   1000.0      0.1    return s1
```

The output of that tells us that `sum_python` takes about 50% of the total time (this varies... run the above cell a few times to get an idea of the average), and `sum_numpy` takes about 10% of the time, and the rest of the time is spent on the `assert` statements.

So let's take a deep dive on the `sum_python` code:

In [15]:
```
%lprun -f sum_python my_sum(1000)
```

```
Timer unit: 1e-09 s

Total time: 0.000937 s
File: /var/folders/40/26f5s8qj42z2mxw1gfrnh5rm0000gn/T/ipykernel_50763/4186063441.py
Function: sum_python at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sum_python(n):
     2         1       2000.0   2000.0      0.2    sum = 0.
     3      1002     496000.0    495.0     52.9    for k in range(n+1):
     4      1001     437000.0    436.6     46.6        sum += k
     5         1       2000.0   2000.0      0.2    return sum
```

The output of this tells is that the line `for k in range(n+1):` takes 40% of the time, and the `sum += k` takes another 60%. This means that the actual loop itself is slow, not the computation in the loop. That's not uncommon for interpreted languages like Matlab and Python (hence the emphasize on "vectorized" code).

We can also take a deep dive on the `sum_numpy` code:

In [16]:
```
%lprun -f sum_numpy my_sum(1000)
```

```
Timer unit: 1e-09 s

Total time: 0.000131 s
File: /var/folders/40/26f5s8qj42z2mxw1gfrnh5rm0000gn/T/ipykernel_50763/4186063441.py
Function: sum_numpy at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           def sum_numpy(n):
     8         1      34000.0  34000.0     26.0    nList = np.arange(n+1)
     9         1      96000.0  96000.0     73.3    sum   = np.sum(nList)
    10         1       1000.0   1000.0      0.8    return sum
```

2b: Your task

Deliverable

Run the code below, and find the **3 most expensive lines of code** from within the scipy.linalg.expm function

In [23]:

```
rng = np.random.default_rng(123456)
A = rng.standard_normal( (3000,3000) )/1e2
eA = sla.expm(A)

%lprun -f sla.expm sla.expm(A)
```

```
Timer unit: 1e-09 s

Total time: 1.39894 s
File: /Library/Frameworks/Python.framework/Versions/3.14/lib/python3.14/site-packages/scipy/linalg/_matfuncs.py
Function: expm at line 234

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   234                                           def expm(A):
   235                                               """Compute the matrix exponential of an array.
   236
   237                                               Array argument(s) of this function may have additional
   238                                               "batch" dimensions prepended to the core shape. In this case, the
array is treated
   239                                               as a batch of lower-dimensional slices; see :ref:`linalg_batch` fo
r details.
   240
   241                                               Parameters
   242                                               ----------
   243                                               A : ndarray
   244                                                   Input with last two dimensions are square ``(..., n, n)``.
   245
   246                                               Returns
   247                                               -------
   248                                               eA : ndarray
   249                                                   The resulting matrix exponential with the same shape of ``A``
   250
   251                                               Notes
   252                                               -----
   253                                               Implements the algorithm given in [1]_, which is essentially a Pad
e
   254                                               approximation with a variable order that is decided based on the a
rray
   255                                               data.
   256
   257                                               For input with size ``n``, the memory usage is in the worst case i
n the
   258                                               order of ``8*(n**2)``. If the input data is not of single and doub
le
   259                                               precision of real and complex dtypes, it is copied to a new array.
   260
   261                                               For cases ``n >= 400``, the exact 1-norm computation cost, breaks
even with
   262                                               1-norm estimation and from that point on the estimation scheme giv
en in
   263                                               [2]_ is used to decide on the approximation order.
   264
   265                                               References
```

```
                                         ----------
                                         .. [1] Awad H. Al-Mohy and Nicholas J. Higham, (2009), "A New Scaling
                                                and Squaring Algorithm for the Matrix Exponential", SIAM J. Matrix
                                                Anal. Appl. 31(3):970-989, :doi:`10.1137/09074721X`

                                         .. [2] Nicholas J. Higham and Francoise Tisseur (2000), "A Block Algorithm
                                                for Matrix 1-Norm Estimation, with an Application to 1-Norm
                                                Pseudospectra." SIAM J. Matrix Anal. Appl. 21(4):1185-1201,
                                                :doi:`10.1137/S0895479899356080`

                                         Examples
                                         --------
                                         >>> import numpy as np
                                         >>> from scipy.linalg import expm, sinm, cosm

                                         Matrix version of the formula exp(0) = 1:

                                         >>> expm(np.zeros((3, 2, 2)))
                                         array([[[1., 0.],
                                                 [0., 1.]],
                                         <BLANKLINE>
                                                [[1., 0.],
                                                 [0., 1.]],
                                         <BLANKLINE>
                                                [[1., 0.],
                                                 [0., 1.]]])

                                         Euler's identity (exp(i*theta) = cos(theta) + i*sin(theta))
                                         applied to a matrix:

                                         >>> a = np.array([[1.0, 2.0], [-1.0, 3.0]])
                                         >>> expm(1j*a)
                                         array([[ 0.42645930+1.89217551j, -2.13721484-0.97811252j],
                                                [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])
                                         >>> cosm(a) + 1j*sinm(a)
                                         array([[ 0.42645930+1.89217551j, -2.13721484-0.97811252j],
                                                [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])

                                         """
      1      11000.0  11000.0      0.0      a = np.asarray(A)
      1       5000.0   5000.0      0.0      if a.size == 1 and a.ndim < 2:
                                                return np.array([[np.exp(a.item())]])

      1       4000.0   4000.0      0.0      if a.ndim < 2:
                                                raise LinAlgError('The input array must be at least two-dimens
```

```
ional')
   311         1       5000.0   5000.0      0.0      if a.shape[-1] != a.shape[-2]:
   312                                                   raise LinAlgError('Last 2 dimensions of the array must be squa
re')
   313
   314                                               # Empty array
   315         1       4000.0   4000.0      0.0      if min(*a.shape) == 0:
   316                                                   dtype = expm(np.eye(2, dtype=a.dtype)).dtype
   317                                                   return np.empty_like(a, dtype=dtype)
   318
   319                                               # Scalar case
   320         1       5000.0   5000.0      0.0      if a.shape[-2:] == (1, 1):
   321                                                   return np.exp(a)
   322
   323         1      18000.0  18000.0      0.0      if not np.issubdtype(a.dtype, np.inexact):
   324                                                   a = a.astype(np.float64)
   325         1       8000.0   8000.0      0.0      elif a.dtype == np.float16:
   326                                                   a = a.astype(np.float32)
   327
   328                                               # An explicit formula for 2x2 case exists (formula (2.2) in [1]_).
However, without
   329                                               # Kahan's method, numerical instabilities can occur (See gh-1958
4). Hence removed
   330                                               # here until we have a more stable implementation.
   331
   332         1       4000.0   4000.0      0.0      n = a.shape[-1]
   333         1      10000.0  10000.0      0.0      eA = np.empty(a.shape, dtype=a.dtype)
   334                                               # working memory to hold intermediate arrays
   335         1       5000.0   5000.0      0.0      Am = np.empty((5, n, n), dtype=a.dtype)
   336
   337                                               # Main loop to go through the slices of an ndarray and passing to
expm
   338         2      18000.0   9000.0      0.0      for ind in product(*[range(x) for x in a.shape[:-2]]):
   339         1       4000.0   4000.0      0.0          aw = a[ind]
   340
   341         1      87000.0  87000.0      0.0          lu = bandwidth(aw)
   342         1       5000.0   5000.0      0.0          if not any(lu):  # a is diagonal?
   343                                                       eA[ind] = np.diag(np.exp(np.diag(aw)))
   344                                                       continue
   345
   346                                                   # Generic/triangular case; copy the slice into scratch and sen
d.
   347                                                   # Am will be mutated by pick_pade_structure
   348                                                   # If s != 0, scaled Am will be returned from pick_pade_structu
re.
   349         1    7285000.0 7.28e+06      0.5          Am[0, :, :] = aw
   350         1  363975000.0 3.64e+08     26.0          m, s = pick_pade_structure(Am)
   351         1      17000.0  17000.0      0.0          if (m < 0):
```

```
   352                                                       raise MemoryError("scipy.linalg.expm could not allocate su
fficient"
   353                                                                         " memory while trying to compute the Pad
e "
   354                                                                         f"structure (error code {m}).")
   355         1  702075000.0 7.02e+08     50.2          info = pade_UV_calc(Am, m)
   356         1      17000.0  17000.0      0.0          if info != 0:
   357                                                       if info <= -11:
   358                                                           # We raise it from failed mallocs; negative LAPACK cod
es > -7
   359                                                           raise MemoryError("scipy.linalg.expm could not allocat
e "
   360                                                                         "sufficient memory while trying to compu
te the "
   361                                                                         f"exponential (error code {info}).")
   362                                                       else:
   363                                                           # LAPACK wrong argument error or exact singularity.
   364                                                           # Neither should happen.
   365                                                           raise RuntimeError("scipy.linalg.expm got an internal
LAPACK "
   366                                                                              "error during the exponential compu
tation "
   367                                                                              f"(error code {info})")
   368         1       8000.0   8000.0      0.0          eAw = Am[0]
   369
   370         1       4000.0   4000.0      0.0          if s != 0:  # squaring needed
   371
   372         1       4000.0   4000.0      0.0              if (lu[1] == 0) or (lu[0] == 0):  # lower/upper triangular
   373                                                           # This branch implements Code Fragment 2.1 of [1]_
   374
   375                                                           diag_aw = np.diag(aw)
   376                                                           # einsum returns a writable view
   377                                                           np.einsum('ii->i', eAw)[:] = np.exp(diag_aw * 2**(-s))
   378                                                           # super/sub diagonal
   379                                                           sd = np.diag(aw, k=-1 if lu[1] == 0 else 1)
   380
   381                                                           for i in range(s-1, -1, -1):
   382                                                               eAw = eAw @ eAw
   383
   384                                                               # diagonal
   385                                                               np.einsum('ii->i', eAw)[:] = np.exp(diag_aw * 2.**
(-i))
   386                                                               exp_sd = _exp_sinch(diag_aw * (2.**(-i))) * (sd *
2**(-i))
   387                                                               if lu[1] == 0:  # lower
   388                                                                   np.einsum('ii->i', eAw[1:, :-1])[:] = exp_sd
   389                                                               else:  # upper
   390                                                                   np.einsum('ii->i', eAw[:-1, 1:])[:] = exp_sd
```

```
   391
   392                                                           else:  # generic
   393         4      43000.0  10750.0      0.0                      for _ in range(s):
   394         3  323507000.0 1.08e+08     23.1                          eAw = eAw @ eAw
   395
   396                                                       # Zero out the entries from np.empty in case of triangular inp
ut
   397         1      33000.0  33000.0      0.0              if (lu[0] == 0) or (lu[1] == 0):
   398                                                           eA[ind] = np.triu(eAw) if lu[0] == 0 else np.tril(eAw)
   399                                                       else:
   400         1    1776000.0 1.78e+06      0.1                  eA[ind] = eAw
   401
   402         1       6000.0   6000.0      0.0          return eA
```

From inspecting the output, these three lines took the longest:

Instruction	Hits	Time	Per hit	% Time
`info = pade_UV_calc(Am, m)`	1	702075000.0	7.02e+08	50.2
`m,s = pick_pade_structure(Am)`	1	363975000.0	3.64e+08	26.0
`eAw = eAw @ eAw`	3	323507000.0	1.08e+08	23.1

If you're curious and wanted to look at even lower level functions, it doesn't always work, because some functions are not implemented in Python, so you can't see their source code.

2c. Alternatives

Instead of doing a formal profiler, you can add in manual timing statements. This is less systematic, but sometimes (due to its simplicity) it's enough. Below are some examples.

Instructions

Go through the examples below

timeit

First, let's use the **timeit** package. You can load this like `immport timeit` and use it explicitly, but in a jupyter/ipython notebook, the easiest way to use it is with **line magics** or **cell magics** (i.e., lines that start with `%` or `%%`, respectively).

The idea of **timeit** is that it runs your code a few times to get a good average value

Below is an example usage of **line magics**

In [24]:
```
n = int(1e4)
%timeit sum_python(n)
%timeit sum_numpy(n)
```

```
266 µs ± 6.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
4.77 µs ± 14.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

... and here's a **cell magic**. It will time the entire cell (so in this case, it doesn't tell us which line was slower)

In [25]:
```
%%timeit
n = int(1e4)
sum_python(n)
sum_numpy(n)
```

```
274 µs ± 5.03 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

You can get fancy if you want, and give it flags. `r` tells is how many times to repeat the full run, and `n` is how many reps per run. (It generaly guesses good values for these based on how fast the program runs)

In [26]:
```
%%timeit -o -r 2 -n 1
# n for number of times to run the code, r to repeat the full runs
sum_numpy(n)

tmStructure = _  # this _ is the result of the previous computation
tmStructure
```

```
41.8 µs ± 22.7 µs per loop (mean ± std. dev. of 2 runs, 1 loop each)
```

Out[26]:
```
<TimeitResult : 41.8 µs ± 22.7 µs per loop (mean ± std. dev. of 2 runs, 1 loop each)>
```

time

We can also use the **time** package, which doesn't do any averaging for us:

In [27]:
```
import time
```

```python
start_time = time.time()  # Record the start time
sum_numpy(n)
end_time = time.time()    # Record the end time

execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")

# or...
start_perf_counter = time.perf_counter()
sum_numpy(n)
end_perf_counter = time.perf_counter()

execution_time_perf = end_perf_counter - start_perf_counter
print(f"Execution time (perf_counter): {execution_time_perf} seconds")
```

```
Execution time: 0.0002799034118652344 seconds
Execution time (perf_counter): 0.00032449999707750976 seconds
```

and like the `timeit` package, the `time` package also has **line magics** and **cell magics**:

In [28]:
```python
# Line magics
%time sum_python(n)
%time sum_numpy(n)
```

```
CPU times: user 477 µs, sys: 215 µs, total: 692 µs
Wall time: 501 µs
CPU times: user 70 µs, sys: 17 µs, total: 87 µs
Wall time: 82 µs
```

Out[28]: `np.int64(50005000)`

In [29]:
```python
%%time
sum_python(n)
sum_numpy(n)
```

```
CPU times: user 454 µs, sys: 98 µs, total: 552 µs
Wall time: 466 µs
```

Out[29]: `np.int64(50005000)`

Deliverables for 2c

None. Just make sure to turn in your work for parts 1 and 2b