EE2703 - Week 8

Niranjan A. Kartha <ee21b095@smail.iitm.ac.in>

April 16, 2023

1 Importing the Required Packages

[1]:
```
import numpy as np
import timeit
import cython
```

[2]:
```
%load_ext Cython
```

---

2 Factorial

Below are various different but equivalent ways to implement a factorial in Python.

[3]:
```
# compute x factorial using recursion
def factorial_recursive(x):
    if x == 0:
        return 1
    return x * factorial_recursive(x - 1)

# compute x factorial using a for-loop
def factorial_for(x):
    prod = 1
    for i in range(1, x + 1):
        prod *= i
    return prod

# compute x factorial using a while-loop
def factorial_while(x):
    prod = 1
    while x > 0:
        prod *= x
        x -= 1
    return prod
```

2.1 Timing

[4]:
```
num = 10
```

[5]:
```
print(factorial_recursive(num))
%timeit factorial_recursive(num)
```

```
3628800
1.74 µs ± 51.7 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[6]:
```
print(factorial_for(num))
%timeit factorial_for(num)
```

```
3628800
853 ns ± 86.3 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[7]:
```
print(factorial_while(num))
%timeit factorial_while(num)
```

```
3628800
906 ns ± 206 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[8]:
```
import math
print(math.factorial(num))
%timeit math.factorial(num)
```

```
3628800
101 ns ± 16.9 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[9]:
```
print(np.math.factorial(num))
%timeit np.math.factorial(num)
```

```
3628800
172 ns ± 5.54 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

---

3 Factorial with Cython

3.0.1 No optimizations

[10]:
```
%%cython --annotate

# compute x factorial using recursion
def factorial_recursive_unopt(x):
    if x == 0:
        return 1
    return x * factorial_recursive_unopt(x - 1)

# compute x factorial using a for-loop
```

```
def factorial_for_unopt(x):
    prod = 1
    for i in range(1, x + 1):
        prod *= i
    return prod

# compute x factorial using a while-loop
def factorial_while_unopt(x):
    prod = 1
    while x > 0:
        prod *= x
        x -= 1
    return prod
```

[10]: <IPython.core.display.HTML object>

[11]:
```
num = 10
```

[12]:
```
print(factorial_recursive_unopt(num))
%timeit factorial_recursive_unopt(num)
```

```
3628800
543 ns ± 72.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[13]:
```
print(factorial_for_unopt(num))
%timeit factorial_for_unopt(num)
```

```
3628800
560 ns ± 27.3 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

[14]:
```
print(factorial_while_unopt(num))
%timeit factorial_while_unopt(num)
```

```
3628800
504 ns ± 52.4 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Observations We can see that we get a 3x performance gain by only executing cython without any optimizaions. A lot of code still requires Python interaction.

3.0.2 With optimizations

[15]:
```
%%cython --annotate

# we need to define the function in C
# recursively calling functions in Cython requires a lot of Python
# interaction
cdef int factorial_c(int x):
```

```
    if x == 0:
        return 1
    return x * factorial_c(x - 1)

# compute x factorial using recursion
def factorial_recursive_opt(int x):
    return factorial_c(x)

# compute x factorial using a for-loop
def factorial_for_opt(int x):
    cdef int prod = 1
    cdef int i
    for i in range(1, x + 1):
        prod *= i
    return prod

# compute x factorial using a while-loop
def factorial_while_opt(int x):
    cdef int prod = 1
    while x > 0:
        prod *= x
        x -= 1
    return prod
```

[15]: <IPython.core.display.HTML object>

[16]:
```
num = 10
```

[17]:
```
print(factorial_recursive_opt(num))
%timeit factorial_recursive_opt(num)
```

```
3628800
95.1 ns ± 7.59 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[18]:
```
print(factorial_for_opt(num))
%timeit factorial_for_opt(num)
```

```
3628800
96.1 ns ± 8 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[19]:
```
print(factorial_while_opt(num))
%timeit factorial_while_opt(num)
```

```
3628800
82.2 ns ± 2.71 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

Observations With optimizations, by annotating types properly, we can improve the performance by 10x, which is faster than the built-in functions.

By annotating types, we can let Cython know that we only care about those specific datatypes – so the compiled code can use functions specific to that datatype, instead of using generalized Python functions that need to worry about different contexts.

---

4 Gaussian elimination

The below code performs Gaussian elimination with partial pivoting.

[20]:
```
# solves Ax = b and returns x
def solve_unopt(A, b):
    n = A.shape[0] # number of unknowns
    B = np.concatenate((A, np.expand_dims(b, axis=1)), axis=1) # augmented␣
 ↪matrix

    # bring B to row echelon form
    for i in range(0, n): # loop through diagonal elements
        # implement partial pivoting
        # find the maximum absolute value in this column
        max_k = i
        for k in range(i + 1, n):
            if np.abs(B[k][i]) > np.abs(B[max_k][i]):
                max_k = k

        if B[max_k][i] == 0:
            raise ZeroDivisionError("unsolvable matrix")

        # swap rows so that the maximum value is our new pivot
        B[[i, max_k]] = B[[max_k, i]]

        # reduce the values below
        for j in range(i + 1, n): # loop through rows below
            B[j,i:n+1] -= (B[j][i] / B[i][i]) * B[i,i:n+1]

    # now find the variables
    for i in range(n - 1, -1, -1): # loop up the matrix
        B[i][n] /= B[i][i]
        B[0:i,n] -= B[i][n] * B[0:i,i]

    return B[0:n,n]
```

Now we can perform a benchmark:

[21]:
```
A = np.random.rand(70, 70).astype('complex128')
b = np.random.rand(70).astype('complex128')
```

[22]:
```
print(np.real(solve_unopt(A, b)))
%timeit solve_unopt(A, b)
```

```
[ 0.28458022  0.41811987  0.35951722 -1.4841086   0.34192831 -1.19323975
 -0.04008448  0.13132137 -1.57022915 -0.11797625  1.87989078  0.56437159
  0.53578896  0.38064578  0.26808468  0.34854613 -0.4990348   0.12836496
 -0.08280893  0.29450409  1.53831444  0.65022882 -1.48333131  1.53756755
 -0.8218627  -1.25331315  0.09986085 -1.01432131 -0.68370082 -1.00954743
  0.29077108 -0.47615478 -0.02043481 -0.13414891 -0.17785674  0.73033622
  0.51250925 -0.06930747 -0.16496159  1.68782398 -0.67432033  0.72367413
 -0.26443808  0.48265339  0.96634908 -1.1190962  -0.47930861 -0.36192433
  0.2597842   1.45662618 -1.08864343 -1.54613425  0.5524382   1.11668571
  2.14526777 -0.59341498 -0.24961004  0.46068611 -1.84999627  0.15989435
 -0.38522271 -1.17295861  0.18339709  0.76691204  0.43848154  1.48656385
 -1.33029683 -0.04862492 -0.64582197  0.07738164]
31.8 ms ± 1.61 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Comparing with the numpy function:

[23]:
```
print(np.real(np.linalg.solve(A, b)))
%timeit np.linalg.solve(A, b)
```

```
[ 0.28458022  0.41811987  0.35951722 -1.4841086   0.34192831 -1.19323975
 -0.04008448  0.13132137 -1.57022915 -0.11797625  1.87989078  0.56437159
  0.53578896  0.38064578  0.26808468  0.34854613 -0.4990348   0.12836496
 -0.08280893  0.29450409  1.53831444  0.65022882 -1.48333131  1.53756755
 -0.8218627  -1.25331315  0.09986085 -1.01432131 -0.68370082 -1.00954743
  0.29077108 -0.47615478 -0.02043481 -0.13414891 -0.17785674  0.73033622
  0.51250925 -0.06930747 -0.16496159  1.68782398 -0.67432033  0.72367413
 -0.26443808  0.48265339  0.96634908 -1.1190962  -0.47930861 -0.36192433
  0.2597842   1.45662618 -1.08864343 -1.54613425  0.5524382   1.11668571
  2.14526777 -0.59341498 -0.24961004  0.46068611 -1.84999627  0.15989435
 -0.38522271 -1.17295861  0.18339709  0.76691204  0.43848154  1.48656385
 -1.33029683 -0.04862492 -0.64582197  0.07738164]
327 µs ± 174 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

---

5 Gaussian Elimination wtih Cython

5.0.1 Without any changes made

We first just run with simple compilation using Cython.

[24]:
```
%%cython --annotate

import numpy as np

# solves Ax = b and returns x
```

```python
def solve_1(A, b):
    n = A.shape[0] # number of unknowns
    B = np.concatenate((A, np.expand_dims(b, axis=1)), axis=1) # augmented␣
 ↪matrix

    # bring B to row echelon form
    for i in range(0, n): # loop through diagonal elements
        # implement partial pivoting
        # find the maximum absolute value in this column
        max_k = i
        for k in range(i + 1, n):
            if np.abs(B[k][i]) > np.abs(B[max_k][i]):
                max_k = k

        if B[max_k][i] == 0:
            raise ZeroDivisionError("unsolvable matrix")

        # swap rows so that the maximum value is our new pivot
        B[[i, max_k]] = B[[max_k, i]]

        # reduce the values below
        for j in range(i + 1, n): # loop through rows below
            B[j,i:n+1] -= (B[j][i] / B[i][i]) * B[i,i:n+1]

    # now find the variables
    for i in range(n - 1, -1, -1): # loop up the matrix
        B[i][n] /= B[i][i]
        B[0:i,n] -= B[i][n] * B[0:i,i]

    return B[0:n,n]
```

[24]: <IPython.core.display.HTML object>

[25]:
```python
print(np.real(solve_1(A, b)))
%timeit solve_1(A, b)
```

```
[ 0.28458022  0.41811987  0.35951722 -1.4841086   0.34192831 -1.19323975
 -0.04008448  0.13132137 -1.57022915 -0.11797625  1.87989078  0.56437159
  0.53578896  0.38064578  0.26808468  0.34854613 -0.4990348   0.12836496
 -0.08280893  0.29450409  1.53831444  0.65022882 -1.48333131  1.53756755
 -0.8218627  -1.25331315  0.09986085 -1.01432131 -0.68370082 -1.00954743
  0.29077108 -0.47615478 -0.02043481 -0.13414891 -0.17785674  0.73033622
  0.51250925 -0.06930747 -0.16496159  1.68782398 -0.67432033  0.72367413
 -0.26443808  0.48265339  0.96634908 -1.1190962  -0.47930861 -0.36192433
  0.2597842   1.45662618 -1.08864343 -1.54613425  0.5524382   1.11668571
  2.14526777 -0.59341498 -0.24961004  0.46068611 -1.84999627  0.15989435
 -0.38522271 -1.17295861  0.18339709  0.76691204  0.43848154  1.48656385
 -1.33029683 -0.04862492 -0.64582197  0.07738164]
```

```
30.8 ms ± 1.76 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

As we can observe, since there is a very large amount of Python interaction, there is no performance gain by just running Cython on the code.

5.0.2 Witih basic type annotation

[26]:
```
%%cython --annotate

import numpy as np

# solves Ax = b and returns x
def solve_2(double complex[:, :] A, double complex[:] b):
    cdef Py_ssize_t n = A.shape[0] # number of unknowns

    cdef Py_ssize_t i, j

    # augmented matrix
    cdef double complex[:, :] B = np.concatenate(
            (A, np.expand_dims(b, axis=1)), axis=1
        )

    cdef Py_ssize_t k, max_k
    cdef double complex temp, ratio

    # bring B to row echelon form
    for i in range(0, n): # loop through diagonal elements
        # implement partial pivoting
        # find the maximum absolute value in this column
        max_k = i

        for k in range(i + 1, n):
            if abs(B[k, i]) > abs(B[max_k, i]):
                max_k = k

        if B[max_k, i] == 0:
            raise ZeroDivisionError("unsolvable matrix")

        # swap rows so that the maximum value is our new pivot
        for j in range(0, n + 1):
            temp = B[i, j]
            B[i, j] = B[max_k, j]
            B[max_k, j] = temp

        # reduce the values below
        for j in range(i + 1, n): # loop through rows below
```

```
            ratio = B[j, i] / B[i, i]

            for k in range(i, n + 1):
                B[j, k] = B[j, k] - ratio * B[i, k]

    # now find the variables
    for i in range(n - 1, -1, -1): # loop up the matrix
        B[i, n] = B[i, n] / B[i, i]
        for j in range(0, i):
            B[j, n] = B[j, n] - B[i, n] * B[j, i]

    return B[0:n,n]
```

[26]: <IPython.core.display.HTML object>

[27]:
```
print(np.real(solve_2(A, b)))
%timeit solve_2(A, b)
```

```
[ 0.28458022  0.41811987  0.35951722 -1.4841086   0.34192831 -1.19323975
 -0.04008448  0.13132137 -1.57022915 -0.11797625  1.87989078  0.56437159
  0.53578896  0.38064578  0.26808468  0.34854613 -0.4990348   0.12836496
 -0.08280893  0.29450409  1.53831444  0.65022882 -1.48333131  1.53756755
 -0.8218627  -1.25331315  0.09986085 -1.01432131 -0.68370082 -1.00954743
  0.29077108 -0.47615478 -0.02043481 -0.13414891 -0.17785674  0.73033622
  0.51250925 -0.06930747 -0.16496159  1.68782398 -0.67432033  0.72367413
 -0.26443808  0.48265339  0.96634908 -1.1190962  -0.47930861 -0.36192433
  0.2597842   1.45662618 -1.08864343 -1.54613425  0.5524382   1.11668571
  2.14526777 -0.59341498 -0.24961004  0.46068611 -1.84999627  0.15989435
 -0.38522271 -1.17295861  0.18339709  0.76691204  0.43848154  1.48656385
 -1.33029683 -0.04862492 -0.64582197  0.07738164]
322 µs ± 31.3 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

We need to forego some numpy conveniences such as scalar-matrix multiplication (we need to do this using a loop), and so the code is longer.

We have used typed memoryviews to pass memory between Python and C without much overhead. This improves our performance by 50x, getting a performance comparable to the built-in numpy function to solve matrices.

However, we can see that the part where we construct the augmented matrix still has a large amount of Python interaction, since it calls a numpy function.

5.0.3 Using our own function for the augmented matrix

[28]:
```
%%cython --annotate

import numpy as np
from cython.view cimport array as cvarray
```

```
# creates an augmented matrix by putting b to the right of A
cdef double complex[:, :] augment(double complex[:, :] A, double complex[:] b):
    cdef Py_ssize_t n = A.shape[0]
    cdef double complex[:, :] B = np.zeros((n, n + 1), dtype=np.complex128)

    cdef Py_ssize_t i, j

    for i in range(0, n):
        for j in range(0, n):
            B[i, j] = A[i, j]
        B[i, n] = b[i]

    return B

# solves Ax = b and returns x
def solve_3(double complex[:, :] A, double complex[:] b):
    cdef Py_ssize_t n = A.shape[0] # number of unknowns

    cdef Py_ssize_t i, j

    # augmented matrix
    cdef double complex[:, :] B = augment(A, b)

    cdef Py_ssize_t k, max_k
    cdef double complex temp, ratio

    # bring B to row echelon form
    for i in range(0, n): # loop through diagonal elements
        # implement partial pivoting
        # find the maximum absolute value in this column
        max_k = i

        for k in range(i + 1, n):
            if abs(B[k, i]) > abs(B[max_k, i]):
                max_k = k

        if B[max_k, i] == 0:
            raise ZeroDivisionError("unsolvable matrix")

        # swap rows so that the maximum value is our new pivot
        for j in range(0, n + 1):
            temp = B[i, j]
            B[i, j] = B[max_k, j]
            B[max_k, j] = temp

        # reduce the values below
```

```
        for j in range(i + 1, n): # loop through rows below
            ratio = B[j, i] / B[i, i]

            for k in range(i, n + 1):
                B[j, k] = B[j, k] - ratio * B[i, k]

    # now find the variables
    for i in range(n - 1, -1, -1): # loop up the matrix
        B[i, n] = B[i, n] / B[i, i]
        for j in range(0, i):
            B[j, n] = B[j, n] - B[i, n] * B[j, i]

    return B[0:n,n]
```

[28]: <IPython.core.display.HTML object>

[29]:
```
print(np.real(solve_3(A, b)))
%timeit solve_3(A, b)
```

```
[ 0.28458022  0.41811987  0.35951722 -1.4841086   0.34192831 -1.19323975
 -0.04008448  0.13132137 -1.57022915 -0.11797625  1.87989078  0.56437159
  0.53578896  0.38064578  0.26808468  0.34854613 -0.4990348   0.12836496
 -0.08280893  0.29450409  1.53831444  0.65022882 -1.48333131  1.53756755
 -0.8218627  -1.25331315  0.09986085 -1.01432131 -0.68370082 -1.00954743
  0.29077108 -0.47615478 -0.02043481 -0.13414891 -0.17785674  0.73033622
  0.51250925 -0.06930747 -0.16496159  1.68782398 -0.67432033  0.72367413
 -0.26443808  0.48265339  0.96634908 -1.1190962  -0.47930861 -0.36192433
  0.2597842   1.45662618 -1.08864343 -1.54613425  0.5524382   1.11668571
  2.14526777 -0.59341498 -0.24961004  0.46068611 -1.84999627  0.15989435
 -0.38522271 -1.17295861  0.18339709  0.76691204  0.43848154  1.48656385
 -1.33029683 -0.04862492 -0.64582197  0.07738164]
336 µs ± 33.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

We have improved our performance further by removing this bottleneck.

We can even further optimize our code by passing some compiler directives to Cython.

5.0.4 Using compiler directives

[30]:
```
%%cython --annotate

import numpy as np
cimport cython

# creates an augmented matrix by putting b to the right of A
@cython.boundscheck(False) # don't check if the index falls outside the array
@cython.wraparound(False) # don't wrap negative indices to the other side of␣
 ↪the array
cdef double complex[:, :] augment(double complex[:, :] A, double complex[:] b):
```

```
    cdef Py_ssize_t n = A.shape[0]
    cdef double complex[:, :] B = np.zeros((n, n + 1), dtype=np.complex128)

    cdef Py_ssize_t i, j

    for i in range(0, n):
        for j in range(0, n):
            B[i, j] = A[i, j]
        B[i, n] = b[i]

    return B

# solves Ax = b and returns x
@cython.boundscheck(False)
@cython.wraparound(False)
@cython.cdivision(True) # don't do zero checks while dividing -- divide raw
def solve(double complex[:, :] A, double complex[:] b):
    cdef Py_ssize_t n = A.shape[0] # number of unknowns
    cdef Py_ssize_t i, j

    # augmented matrix
    cdef double complex[:, :] B = augment(A, b)

    cdef Py_ssize_t k, max_k
    cdef double complex temp, ratio

    # bring B to row echelon form
    for i in range(0, n): # loop through diagonal elements
        # implement partial pivoting
        # find the maximum absolute value in this column
        max_k = i

        for k in range(i + 1, n):
            if abs(B[k, i]) > abs(B[max_k, i]):
                max_k = k

        if B[max_k, i] == 0:
            raise ZeroDivisionError("unsolvable matrix")

        # swap rows so that the maximum value is our new pivot
        for j in range(0, n + 1):
            temp = B[i, j]
            B[i, j] = B[max_k, j]
            B[max_k, j] = temp

        # reduce the values below
        for j in range(i + 1, n): # loop through rows below
```

```
            ratio = B[j, i] / B[i, i]

            for k in range(i, n + 1):
                B[j, k] = B[j, k] - ratio * B[i, k]

    # now find the variables
    for i in range(n - 1, -1, -1): # loop up the matrix
        B[i, n] = B[i, n] / B[i, i]
        for j in range(0, i):
            B[j, n] = B[j, n] - B[i, n] * B[j, i]

    return B[0:n,n]
```

[30]: <IPython.core.display.HTML object>

[31]:
```
print(np.real(solve(A, b)))
%timeit solve(A, b)
```

```
[ 0.28458022  0.41811987  0.35951722 -1.4841086   0.34192831 -1.19323975
 -0.04008448  0.13132137 -1.57022915 -0.11797625  1.87989078  0.56437159
  0.53578896  0.38064578  0.26808468  0.34854613 -0.4990348   0.12836496
 -0.08280893  0.29450409  1.53831444  0.65022882 -1.48333131  1.53756755
 -0.8218627  -1.25331315  0.09986085 -1.01432131 -0.68370082 -1.00954743
  0.29077108 -0.47615478 -0.02043481 -0.13414891 -0.17785674  0.73033622
  0.51250925 -0.06930747 -0.16496159  1.68782398 -0.67432033  0.72367413
 -0.26443808  0.48265339  0.96634908 -1.1190962  -0.47930861 -0.36192433
  0.2597842   1.45662618 -1.08864343 -1.54613425  0.5524382   1.11668571
  2.14526777 -0.59341498 -0.24961004  0.46068611 -1.84999627  0.15989435
 -0.38522271 -1.17295861  0.18339709  0.76691204  0.43848154  1.48656385
 -1.33029683 -0.04862492 -0.64582197  0.07738164]
276 µs ± 29 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Our functions now are running almost entirely in C, and we have a better performance than all of the above ones.

6 SPICE – unchanged

Since the bottlenecks in the SPICE solver are related to file operations, it does not make much sense to optimize the Python code.

Therefore, the below code is unchanged compared to assignment 2.

The below code blocks set up a `SpiceSolver` class which can be used to load and solve SPICE netlists.

6.1 General class definitions

[32]:
```python
# an edge in the network that goes between two nodes
class Edge:
    def __init__(self, name, node_left, node_right):
        self.name = name
        self.node_left = node_left
        self.node_right = node_right

# a general passive component
class Passive(Edge):
    def __init__(self, name, node_left, node_right, value, cond=0):
        Edge.__init__(self, name, node_left, node_right)
        self.value = value
        self.cond = cond # condition, like voltage of a capacitor

# a general source (voltage or current)
class Source(Edge):
    def __init__(self, name, node_left, node_right, value, phase):
        Edge.__init__(self, name, node_left, node_right)
        self.value = value
        self.phase = phase
```

6.2 Spice Solver class

[33]:
```python
# class that parses a spice file and solves it
class SpiceSolver:
    # throw an error specifying the line where the error occurred
    @staticmethod
    def __spice_err(line_index, line, message):
        raise Exception(
            f"SPICE ERROR on line {line_index + 1}:\n" +
            f"{line}\n" +
            message
        )

    # parse a float, and throw an error if it's not valid
    @staticmethod
    def __parse_float(x, line_index, line):
        try:
            x = float(x)
        except ValueError:
            SpiceSolver.__spice_err(line_index, line, f"couldn't parse as␣
↪number: `{command[3]}`")
        return x

    # validate the number of arguments to a command
```

```python
    @staticmethod
    def __assert_arg_count(command, expected, line_index, line):
        if len(command) not in expected:
            SpiceSolver.__spice_err(line_index, line, f"invalid number of arguments: expected {expected}, got {len(command)}")

    def __init__(self):
        # we will replace all nodes with numbers in our matrix
        # this map will remember the mapping between node names and the numbering we assign
        self.node_map = { "GND": 0 }

        # number of nodes (excluding ground)
        self.node_count = 0

        # lists of sources by frequency
        self.voltages = { 0: [] }
        self.currents = { 0: [] }

        # store a list of components
        self.resistors = []
        self.capacitors = []
        self.inductors = []

        # we represent the passive elements in the circuit as 3 weighted graphs where
        # components form edges:
        # one has conductances as its weights
        # one has capacitance as its weights
        # one has 1/inductance as its weights

        # we use the diagonal elements as the negative of the sum of the rest of the
        # values in the corresponding row (plus ground), since this is what appears
        # in the MNA matrix

        # adjacency matrix of conductances between nodes (excluding GND)
        self.conductance_matrix = None

        # adjacency matrix of capacitances between nodes (excluding GND)
        self.capacitance_matrix = None

        # adjacency matrix of 1/inductances between nodes (excluding GND)
        self.inv_inductance_matrix = None

    # read a file and create a representation of the circuit in memory
```

```
    def read_file(self, filename):
        # open file
        with open(filename) as f:
            l = f.readlines()

            if len(l) == 0: # make sure the file is not empty
                raise Exception("SPICE ERROR: empty file")

            # seek to start of circuit
            start = 0
            while len(l[start]) > 0 and l[start].split("#")[0].strip() != ".
↪circuit":
                start += 1
                if start >= len(l): # we need to have a ".circuit" somewhere
                    raise Exception("SPICE ERROR: couldn't find start of␣
↪circuit")

            # find end of circuit
            end = start
            while len(l[end]) > 0 and l[end].split("#")[0].strip() != ".end":
                end += 1
                if end >= len(l): # we need to have a ".end" somewhere
                    raise Exception("SPICE ERROR: couldn't find end of circuit")

            # generate a dictionary of source names to frequencies based on the␣
↪directives given after ".end"
            frequencies = {}

            # loop through lines after the ".end" directive
            for i in range(end + 1, len(l)):
                command = l[i].split('#')[0].split() # get command

                if command[0] == "": # if there's no command, move on
                    continue

                # look for and parse ".ac" command
                if command[0].lower() == ".ac":
                    self.__assert_arg_count(command, [3], i, l[i])

                    if command[1].upper() in frequencies: # each source should␣
↪have its frequency defined only once
                        self.__spice_error(i, l[i], "redefinition of freqency")

                    # add frequency to dictionary
                    freq = self.__parse_float(command[2], i, l[i])
                    frequencies[command[1].upper()] = freq
```

```
                self.voltages[freq] = []
                self.currents[freq] = []
            else:
                pass # we don't need to throw errors if there is garbage after the ".end"

        # generate a list of nodes and components, and check syntax
        for i in range(start + 1, end): # loop through the ".circuit" section
            command = l[i].split('#')[0].split()

            if command[0] == "":
                continue

            if len(command) < 4: # all commands have at least 4 arguments
                self.__spice_error(i, l[i], "invalid number of arguments")

            # the second and third arguments will contain node names
            # add the nodes mentioned to our node map if they aren't already in it
            for j in [1, 2]:
                node_name = command[j]
                if node_name.upper() not in self.node_map:
                    self.node_count += 1
                    self.node_map[node_name.upper()] = self.node_count

            # type of component
            ctype = l[i][0].upper()

            # parse passive components
            if ctype in ["R", "L", "C"]:
                self.__assert_arg_count(command, [4, 5], i, l[i])

                edge = Passive(
                    command[0],
                    self.node_map[command[1].upper()],
                    self.node_map[command[2].upper()],
                    self.__parse_float(command[3], i, l[i])
                )

                if len(command) == 5: # if an initial condition is specified (5th argument), set it
                    edge.cond = self.__parse_float(command[4], i, l[i])

                # add value to list
                if ctype == "R":
                    self.resistors.append(edge)
```

```python
                    elif ctype == "L":
                        self.inductors.append(edge)
                    else:
                        self.capacitors.append(edge)

                # parse sources
                elif ctype in ["V", "I"]:
                    freq = 0
                    phase = 0

                    # check type of source
                    if command[3].lower() == "ac":
                        self.__assert_arg_count(command, [6], i, l[i])

                        # look up our frequencies dictionary for this source
                        if command[0].upper() in frequencies:
                            freq = frequencies[command[0].upper()]
                            phase = self.__parse_float(command[5], i, l[i])
                        else: # throw an error if the frequency for this source is not defined
                            self.__spice_error(i, l[i], f"could not find frequency for AC source: `{command[0]}`")

                    elif command[3].lower() == "dc":
                        self.__assert_arg_count(command, [5], i, l[i])

                    else:
                        self.__spice_error(i, l[i], f"invalid source type: `{command[3]}`")

                    edge = Source(
                        command[0],
                        self.node_map[command[1].upper()],
                        self.node_map[command[2].upper()],
                        self.__parse_float(command[4], i, l[i]),
                        phase,
                    )

                    if ctype == "V":
                        self.voltages[freq].append(edge)
                    else:
                        self.currents[freq].append(edge)

                else:
                    self.__spice_error(i, l[i], "unidentified command inside circuit")
```

```
        # delete GND
        del self.node_map["GND"]

        # now use the data we have parsed to generate matrices
        self.__gen_matrices()

    # generate conductance, capacitance and inverse-inductance matrices
    def __gen_matrices(self):
        # initialize to zeroes
        # outwards currents are taken as positive

        self.conductance_matrix = np.zeros((self.node_count, self.node_count))
        self.capacitance_matrix = np.zeros((self.node_count, self.node_count))
        self.inv_inductance_matrix = np.zeros((self.node_count, self.
↪node_count))

        # conductance matrix
        for r in self.resistors:
            low = min(r.node_left, r.node_right) - 1
            high = max(r.node_left, r.node_right) - 1
            if low == -1:
                # if the resistor is between ground and a node, add it to the␣
↪diagonal
                self.conductance_matrix[high][high] += 1 / r.value
            else:
                self.conductance_matrix[low][high] -= 1 / r.value

        # capacitance matrix
        for c in self.capacitors:
            low = min(c.node_left, c.node_right) - 1
            high = max(c.node_left, c.node_right) - 1
            if low == -1:
                self.capacitance_matrix[high][high] += c.value
            else:
                self.capacitance_matrix[low][high] -= c.value

        # inverse-inductance matrix
        for l in self.inductors:
            low = min(l.node_left, l.node_right) - 1
            high = max(l.node_left, l.node_right) - 1
            if low == -1:
                self.inv_inductance_matrix[high][high] += 1 / l.value
            else:
                self.inv_inductance_matrix[low][high] -= 1 / l.value

        # mirror our matrices and fix the diagonal elements
```

```
        mats = [self.conductance_matrix, self.inv_inductance_matrix, self.
↪capacitance_matrix]
        for mat in mats:
            for i in range(self.node_count):
                # set diagonal elements
                for j in range(0, i):
                    mat[i][i] -= mat[j][i]

                # mirror
                for j in range(i + 1, self.node_count):
                    mat[i][i] -= mat[i][j]
                    mat[j][i] = mat[i][j]

    # create a linear matrix equation A x = b for MNA (returns A and b)
    def __gen_mna_pair(self, modified_voltages, freq):
        w = 2j * np.pi * freq

        dim = self.node_count + len(modified_voltages) # number of dimensions␣
↪in matrix

        mat = np.zeros((dim, dim), dtype=np.complex_)
        x = np.zeros(dim, dtype=np.complex_)


        if w != 0:
            mat[0:self.node_count, 0:self.node_count] = self.conductance_matrix␣
↪+ \
                                                                    1j * w * self.
↪capacitance_matrix + \
                                                                    1 / (1j * w) * self.
↪inv_inductance_matrix
        else:
            mat[0:self.node_count, 0:self.node_count] = self.conductance_matrix

        # outward currents are taken as positive, active sign convention is␣
↪followed for voltage sources

        k = self.node_count

        # create MNA matrix

        # add equations of the form V_left - V_right = V_source
        # and also currents on the RHS of KCL equations
        for v in modified_voltages:
            if v.node_left != 0:
                mat[v.node_left - 1][k] = -1 # current
```

```
                mat[k][v.node_left - 1] = 1 # voltage

            if v.node_right != 0:
                mat[v.node_right - 1][k] = 1 # current
                mat[k][v.node_right - 1] = -1 # voltage

            x[k] = v.value * np.exp(1j * v.phase)

            k += 1

        for i in self.currents[freq]:
            if i.node_left != 0:
                x[i.node_left - 1] = i.value * (np.exp(1j * i.phase))

            if i.node_right != 0:
                x[i.node_right - 1] = -i.value * (np.exp(1j * i.phase))

        return (mat, x)

    # generate modified voltage sources according to frequency and solve,
    # returning a dictionary of nodes to voltages/currents
    def __solve_freq(self, freq):
        modified_voltages = []

        # map of extra nodes that we have added
        extra_nodes = {}

        # convert inductors to 0-voltage sources for DC
        if freq == 0:
            for l in self.inductors:
                modified_voltages.append(
                    Source(l.name, l.node_left, l.node_right, 0, 0)
                )

        for f in self.voltages:
            # if the source is at a different frequency, short it
            if freq != f:
                for v in self.voltages[f]:
                    modified_voltages.append(
                        Source(v.name, v.node_left, v.node_right, 0, 0)
                    )
            else:
                for v in self.voltages[f]:
                    modified_voltages.append(v)

        # keep track of where each voltage is
        new_node_count = self.node_count
```

```
        for v in modified_voltages:
            new_node_count += 1
            extra_nodes[f"I_{v.name}"] = new_node_count

        solution = solve(*self.__gen_mna_pair(modified_voltages, freq))

        if freq == 0:
            solution = np.real(solution)

        result = {}

        for key in self.node_map:
            result[f"V_{key}"] = solution[self.node_map[key] - 1]

        for key in extra_nodes:
            result[key] = solution[extra_nodes[key] - 1]

        return result

    # print the steady state solution of the system
    def solve_steady(self):
        for freq in self.voltages:
            print(f"frequency {freq}:")
            print()
            try:
                sol = self.__solve_freq(freq)
                for key in sol:
                    print(f"{key:<10}{sol[key]}")
            except ZeroDivisionError:
                print("no steady state at this frequency")

            print()
            print("=" * 20)
            print()
```

We can invoke the solver as follows:

It prints the different responses of the circuit to the different frequencies of sources that are present.

[34]:
```
solver = SpiceSolver()
solver.read_file("spice/ckt6.netlist")
solver.solve_steady()
```

```
frequency 0:

V_N3      -0.0
V_N1      0.0
V_N2      0.0
I_L1      -0.0
```

```
I_V1      0.0

====================

frequency 1000.0:

V_N3      (-5-0j)
V_N1      (3.141612892928987e-05-0j)
V_N2      (3.221190877143386e-05-0j)
I_V1      (0.0050000322119087715+0j)

====================
```