# 1

Monday, January 20, 2025 6:42 PM

prob1

# prob1

January 20, 2025

[10]:
```
import concurrent.futures
import time

# run with cuda
import cupy as cp
import cupyx.scipy.sparse as sp
import cupyx.scipy.sparse.linalg as splinalg
cp.get_default_memory_pool().free_all_blocks()

# for running without cuda
# import numpy as cp
# import scipy.sparse as sp
# import scipy.sparse.linalg as splinalg
```

[11]:
```
def timed_problem(problem, matrix_size):
    A, b, solver = problem(matrix_size)
    # we aren't counting the time it takes to construct the matrix
    start_time = time.time()
    solver(A, b)
    end_time = time.time()
    cp.get_default_memory_pool().free_all_blocks()
    return end_time - start_time


def run_problems(
    problems: list, matrix_sizes=[10**i for i in range(1, 5)], timeout=60 * 5
):
    run_times = {
        problem.__name__: {matrix_size: None for matrix_size in matrix_sizes}
        for problem in problems
    }

    for problem in problems:
        print(f"Running {problem.__name__} \n")
        # I don't want the problems to run in parallel
        with concurrent.futures.ThreadPoolExecutor(max_workers=1) as executor:
            futures = {
```

```
                matrix_size: executor.submit(timed_problem, problem,␣
↪matrix_size)
                for matrix_size in matrix_sizes
            }

            for future in concurrent.futures.as_completed(futures.values()):
                matrix_size = next(
                    key for key, value in futures.items() if value == future
                )
                matrix_size_text = f"{matrix_size:.0e}"
                try:
                    run_time = future.result(timeout=timeout)
                    print(f"Finished for matrix size {matrix_size_text} in␣
↪{run_time} seconds")
                    run_times[problem.__name__][matrix_size] = run_time
                except concurrent.futures.TimeoutError:
                    print(f"Timeout for matrix size {matrix_size_text}")
                    break
                except Exception as e:
                    print(f"Exception for matrix size {matrix_size_text}: {e}")
                    break

    return run_times

run_times = {}
```

[12]:
```
def prob1(matrix_size):
    """
    1. Diagonal matrix: for N  [10, 109] in factors of 10 until compute time
    seems unreasonable.
    • Lx and Ux are 0
    • D0 to DN and b0 to bN are 1

    This problem can be solved algebraically. Di = bi for all i.
    """
    A = sp.eye(matrix_size, format="csr")
    b = cp.ones(matrix_size)
    return A, b, splinalg.spsolve_triangular

run_times.update(run_problems([prob1], matrix_sizes=[10**i for i in range(1,␣
↪9)]))
```

```
Running prob1

Finished for matrix size 1e+01 in 0.0 seconds
Finished for matrix size 1e+02 in 0.007524013519287109 seconds
Finished for matrix size 1e+03 in 0.0010099411010742188 seconds
Finished for matrix size 1e+04 in 0.0009906291961669922 seconds
```

```
Finished for matrix size 1e+05 in 0.0014879703521728516 seconds
Finished for matrix size 1e+06 in 0.008454561233520508 seconds
Finished for matrix size 1e+07 in 0.22781896591186523 seconds
Finished for matrix size 1e+08 in 0.36722874641418457 seconds
```

[13]:
```python
def prob2(matrix_size):
    """
    2. Lower triangular matrix: for N  [10, 109] in factors of 10 until compute
    time seems unreasonable.
    • U1 to UN are 0
    • LA is -1/A for A  [1, N ]
    • D1 to DN is one minus the sum of LA in the row
    • b0 to bN are 1
    """
    # Create diagonal values
    diag_values = {i: -1 / i for i in range(1, matrix_size)}

    # Create diagonals
    diags = {i: cp.full((matrix_size - i), value) for i, value in diag_values.
↪items()}
    diags[0] = cp.cumsum(cp.array([1] + [-1 * i for i in diag_values.values()]))

    # Create the lower triangular matrix
    mat = sp.tril(
        sp.diags(
            list(diags.values()),
            -1 * cp.array(list(diags.keys())),
            format="csr",
            dtype=cp.float32,
        )
    )

    b = cp.ones(matrix_size, dtype=cp.float32)

    return mat, b, splinalg.spsolve_triangular

prob2_results = run_problems([prob2])
run_times.update(prob2_results)
```

```
Running prob2

Finished for matrix size 1e+01 in 0.010002613067626953 seconds
Finished for matrix size 1e+02 in 0.0284731388092041 seconds
Finished for matrix size 1e+03 in 0.030472993850708008 seconds
Finished for matrix size 1e+04 in 4.286558389663696 seconds
```

[14]:
```python
def prob3(matrix_size):
    """
```

```
    3. Upper-Triangular matrix: for N  [10, 109] in factors of 10 until compute
    time seems unreasonable.
    • L1 to LN are 0
    • UA is -1/A for A  [1, N ]
    • D1 to DN is one minus the sum of UA in the row
    • b0 to bN are 1
    """
    # Create diagonal values
    diag_values = {i: -1 / i for i in range(1, matrix_size)}

    # Create diagonals
    diags = {i: cp.full((matrix_size - i), value) for i, value in diag_values.
↪items()}
    diags[0] = cp.cumsum(cp.array([1] + [-1 * i for i in diag_values.
↪values()]))[::-1]

    # Create the lower triangular matrix
    mat = sp.diags(
        list(diags.values()),
        1 * cp.array(list(diags.keys())),
        format="csr",
        dtype=cp.float32,
    )

    b = cp.ones(matrix_size, dtype=cp.float32)

    return mat, b, splinalg.spsolve_triangular


prob3_results = run_problems([prob3])
run_times.update(prob3_results)
```

```
Running prob3

Finished for matrix size 1e+01 in 0.0005018711090087891 seconds
Finished for matrix size 1e+02 in 0.0 seconds
Finished for matrix size 1e+03 in 0.0010006427764892578 seconds
Finished for matrix size 1e+04 in 0.004502773284912109 seconds
```

[15]:
```
def prob4(matrix_size):
    """
    4. Tridiagonal matrix: for N  [10, 109] in factors of 10 until compute time
    seems unreasonable.
    • L1 to LN are -1
    • U1 to UN are -1
    • D1 to DN are 3
    • b0 to bN are 1
```

```
    """
    value_map = {-1 : -1, 0 : 3, 1 : -1}
    mat = sp.diags(
        list(value_map.values()),
        list(value_map.keys()),
        shape=(matrix_size, matrix_size),
        format="csr",
    )
    b = cp.ones(matrix_size)

    return mat, b, splinalg.spsolve

prob4_results = run_problems([prob4], matrix_sizes=[10**i for i in range(1, 7)])
run_times.update(prob4_results)
```

```
Running prob4

Finished for matrix size 1e+01 in 0.005156993865966797 seconds
Finished for matrix size 1e+02 in 0.0016646385192871094 seconds
Finished for matrix size 1e+03 in 0.01132059097290039 seconds
Finished for matrix size 1e+04 in 0.10631823539733887 seconds
Finished for matrix size 1e+05 in 1.092623233795166 seconds
Finished for matrix size 1e+06 in 11.00407862663269 seconds
```

[16]:
```
def prob5(matrix_size):
    """
    5. Banded matrix: for N  [10, 109] in factors of 10 until compute time␣
 ↪seems
    unreasonable.
    • U1, U5, L1, and L5 are -1
    • D1 to DN is 5
    • b0 to bN are 1
    • The rest of U and L are zero.
    """
    value_map = {-1: -1, -5 : -1, 1 : -1, 5 : -1, 0 : 5}
    mat = sp.diags(
        list(value_map.values()),
        list(value_map.keys()),
        shape=(matrix_size, matrix_size),
        format="csr",
    )
    b = cp.ones(matrix_size)

    return mat, b, splinalg.spsolve


prob5_results = run_problems([prob5], matrix_sizes=[10**i for i in range(1, 7)])
```

```
run_times.update(prob5_results)
```

```
Running prob5

Finished for matrix size 1e+01 in 0.001978635787963867 seconds
Finished for matrix size 1e+02 in 0.002065420150756836 seconds
Finished for matrix size 1e+03 in 0.01174616813659668 seconds
Finished for matrix size 1e+04 in 0.11741328239440918 seconds
Finished for matrix size 1e+05 in 1.1982793807983398 seconds
Finished for matrix size 1e+06 in 11.911452054977417 seconds
```

[18]:
```
def prob6(matrix_size):
    """
    7. Upwind matrix: for N  [10, 109] in factors of 10 until compute time␣
 ↪seems
    unreasonable.
    • U1 to UN and L2 to LN are 0
    • L1 = -0.9
    • D1 to DN is 1
    • b1 to bN are 0, but b0 = 1
    """
    value_map = {-1: -0.9, 0 : 1}
    A = sp.diags(
        list(value_map.values()),
        list(value_map.keys()),
        shape=(matrix_size, matrix_size),
        format="csr",
    )
    b = cp.zeros(matrix_size)
    b[0] = 1

    return A, b, splinalg.spsolve

prob6_results = run_problems([prob6], matrix_sizes=[10**i for i in range(1, 8)])
run_times.update(prob6_results)
```

```
Running prob6

Finished for matrix size 1e+01 in 0.010062694549560547 seconds
Finished for matrix size 1e+02 in 0.017606019973754883 seconds
Finished for matrix size 1e+03 in 0.10295820236206055 seconds
Finished for matrix size 1e+04 in 0.277296781539917 seconds
Finished for matrix size 1e+05 in 1.1173458099365234 seconds
Finished for matrix size 1e+06 in 10.917600870132446 seconds
Finished for matrix size 1e+07 in 179.83606696128845 seconds
```

1. What did you learn about your computers limitations

   I learned that the ability to solve the matrix isn't the main problem. Memory is the main problem, followed by some speed issues in matrix construction.

2. Were there algebraic simplifications you used that made some problems dramatically easier than others?

   I didn't use any to make the assessment fair, but there were simplifications for all of them. For the banded matrices, I could have solved much larger matrices by writing my own solver and storing only the value in the band itself.

   A similar method could be applied to triangular matrices and they would be easier to solve.

3. Were there matrix-math tools that helped with some problems, but not others?

   I was able to use a triangular matrix solver for the triangular matrices. All matrices were sparse, so I could use the sparse solvers. But in the end, the main issue was memory so the matrix math tools were kind of irrelevant.

4. Did you try different coding languages/tools that made some easier than others?

   No. CPP and FORTRAN could be faster because of less overhead, but I'm just using a CPP/CUDA wrapper here, so it is as fast as it is going to get. I tried running these problems on my CPU instead of my GPU. My CPU has more RAM than my GPU has VRAM, but the amount of memory for these matrices goes up exponentially, so it didn't really change my ability to solve the large matrices. The CPU was much slower.

2

Monday, January 20, 2025 6:51 PM

View factors and Transport connection

1. If there is an infinitely large electrically heated plate that is 6cm from a parallel infinitely large cold plate, what fraction of the heat from the hot plate can *see* the cold plate?
2. If that hot plate is only 1 $cm^2$, what fraction can *see* the cold plate?
3. If the hot and cold plates were instead very thin wires, that are 6cm apart what fraction of the heat from the hot wire can *see* the cold wire?
4. if the cold and hot plate were only 1 cm square on each side (and 6cm apart), how would you compute the fraction can *see* the cold plate?
5. As the cold plate moves closer/further from the hot plate, does the fraction change?

1) 1/2 of the heat can see the cold plate

2) 

Still, only $\frac{1}{2}$ of the hot plate can see the cold plate.

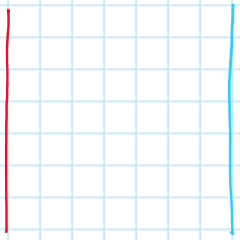
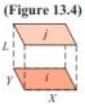
3) 

**TABLE 13.2 View Factors for Three-Dimensional Geometries [4]**

| Geometry | Relation |
|---|---|
| **Aligned Parallel Rectangles** (Figure 13.4)  | $\bar{X} = X/L, \bar{Y} = Y/L$ C1) $F_{ij} = \frac{2}{\pi \bar{X} \bar{Y}} \left\{ \ln \left[ \frac{(1+\bar{X}^2)(1+\bar{Y}^2)}{1+\bar{X}^2+\bar{Y}^2} \right]^{1/2} + \bar{X}(1+\bar{Y}^2)^{1/2} \tan^{-1} \frac{\bar{X}}{(1+\bar{Y}^2)^{1/2}} + \bar{Y}(1+\bar{X}^2)^{1/2} \tan^{-1} \frac{\bar{Y}}{(1+\bar{X}^2)^{1/2}} - \bar{X} \tan^{-1} \bar{X} - \bar{Y} \tan^{-1} \bar{Y} \right\}$ |

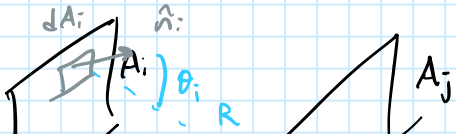
As the wires become infintisemly thin, they can see less and less of each other.

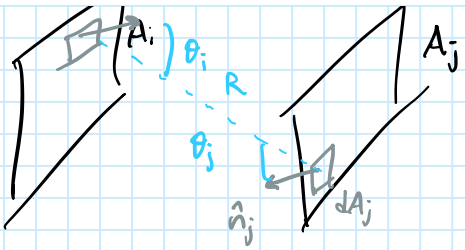
This can be proven using the above equation. C1)

As $\bar{X} \to \infty$ and $\bar{Y} \to 0$ (or vice versa), $F_{ij} \to 0$

4) I would use Equation C1) from DeWitt Copen Heat Transfer

But...






Find the solid angle subtended by $A_j$ from $dA_i$, which is determined by integrating $dA_j$ over $A_j$.

$$F_{ij} = \frac{1}{A_i}\int_{A_i}\int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi R^2} dA_i dA_j \qquad (2)$$

You have to convert the coordinates to $X$, $Y$, which is extremely painful.

5) Intuitively,

as the plates get closer, the view factor increases

as the plates get further, the view factor decreases

Based on (2)

$F_{ij} \propto \frac{1}{R^2}$ if $R \uparrow$, $F_{ij} \downarrow$

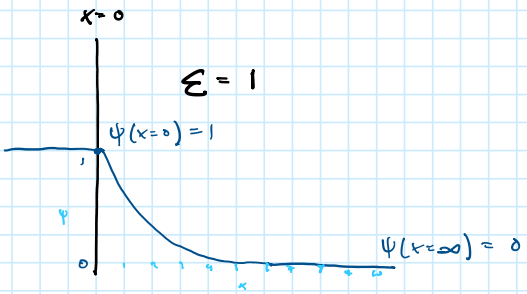
if $R \downarrow$, $F_{ij} \uparrow$

# 3

Monday, January 20, 2025 7:29 PM

There is a beam of neutrons with an incident angular flux of one that is parallel with the x-axis and perpendicular to the left-surface of a purely-absorbing material with a macroscopic cross section of $\Sigma = 1$ reactions per cm of neutron travel. The transport equation for this simplified problem is $\frac{d\psi}{dx} + \Sigma_t \psi = 0$, where $\psi(x = 0) = 1$.

1. If you need to know the angular flux as a function of distance from the left face, you can solve for this equation analytically. Write the equation and plot the solution for $x \in [0, 10]$ cm.
2. If you want the analytic solution at every 0.1 cm interval, you could solve for all 100 points (nodes) in space and plot the solution on the same graph.
3. If you know the solution on the node to the left of any given node, then you can create an algebraic expression for the analytic solution for the new node (e.g. using the solution at x=0 to compute the solution at x=0.1). This would create a series of solutions each depend on the previous node. This problem could be put in matrix form. Compare it with the upwind problem previously solved.




1) $\frac{d\psi}{dx} + \Sigma_t \psi = 0$

$\int \frac{1}{\psi} d\psi = \int -\Sigma_t \, dx$

$\ln \psi = -\Sigma_t x + C$

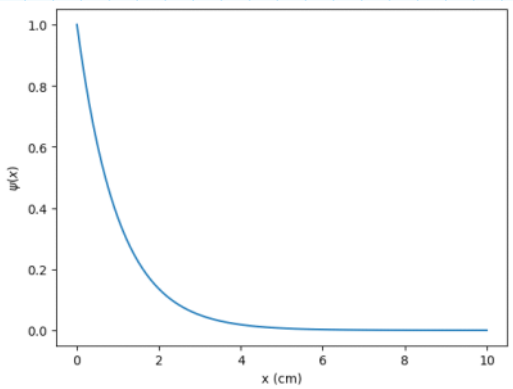
$\psi = C_0 \cdot \exp(-\Sigma_t x)$

$\psi(0) = 1 = C_0 \cdot \exp(-\Sigma_t \cdot 0)$

$1 = C_0$

$\psi = \exp(-\Sigma_t x)$

2) This is how computers plot anyways



3) $\frac{d\psi}{dx} + \varepsilon_t \psi = 0$

$\frac{d\psi}{dx} = -\varepsilon_t \psi$

$\psi_{i+1} = \psi_i + \left.\frac{d\psi}{dx}\right|_{x_i} \Delta x$

$\psi_{i+1} = \psi_i - \varepsilon_t \psi_i \Delta x = \psi_i (1 - \varepsilon_t \Delta x)$

where $\psi_{i=0} = 1$ and $x_{i=0} = 0$

For 3 steps:

$\psi_0 = \psi_{initial}$

$\psi_1 - \psi_0 (1 - \varepsilon_t \Delta x) = 0$

$\psi_2 - \psi_1 (1 - \varepsilon_t \Delta x) = 0$

Let $L = 1 - \varepsilon_t \Delta x$

$$\begin{bmatrix} 1 & 0 & 0 \\ -L & 1 & 0 \\ 0 & -L & 1 \end{bmatrix} \begin{bmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

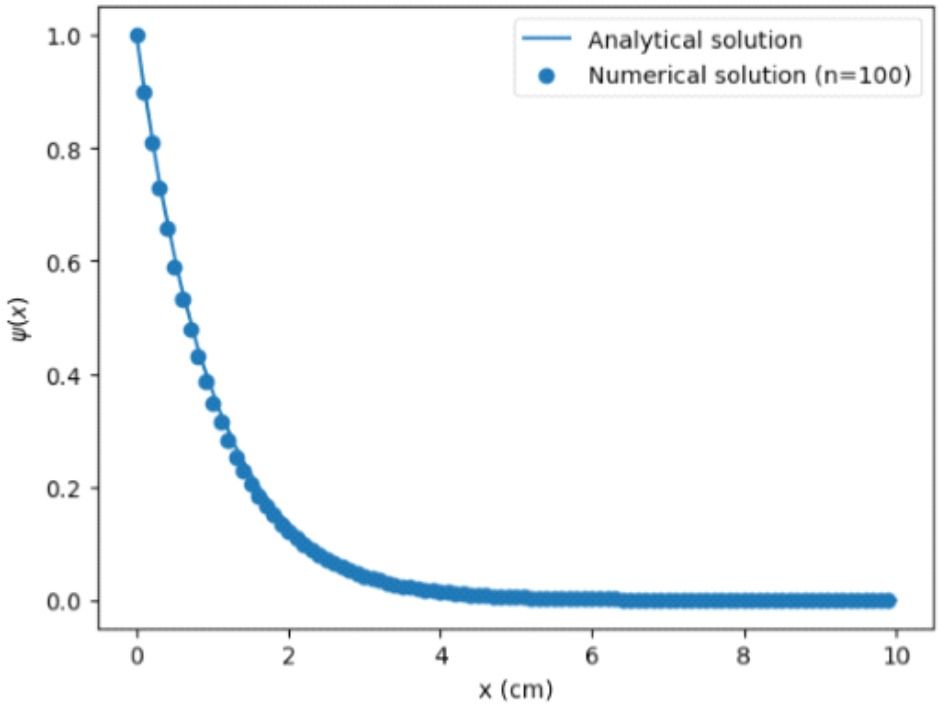
prob3

# prob3

January 20, 2025

[56]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

[57]:
```
x = np.linspace(0, 10, 101)
analytical_solution = lambda x : np.exp(-x)

fig, ax = plt.subplots()
ax.plot(x, analytical_solution(x), label='Analytical solution')

ax.set_xlabel('x (cm)')
ax.set_ylabel(r"$\psi(x)$")

##

initial_value = 1
cross_section = 1

step = 0.1
n = 100

A = sp.sparse.diags([1, cross_section * step - 1], [0, -1], shape=(n, n),␣
 ↪format='csc')
b = [initial_value] + [0 for _ in range(n - 1)]
solution = sp.sparse.linalg.spsolve(A, b)


x = [i * step for i in range(n)]
ax.scatter(x, solution, label="Numerical solution (n=100)")
ax.legend();
```


Analytical solution
Numerical solution (n=100)
1.0
0.8
0.6
0.4
0.2
0.0
ψ(x)
0
2
4
6
8
10
x (cm)

[ ]: