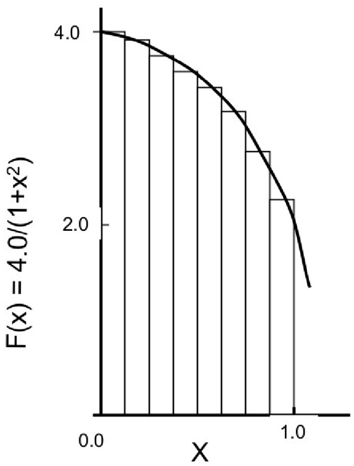
Exercise 5

Task 1 (`pi_seq.c` and `pi_par.c`)

In this exercise, you will first implement a **sequential** approximation of π, and then develop your **first parallel OpenMP version**.

Task 1b: Sequential Version

Implement the following program that computes π using numerical integration:



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

```
static long num_steps = 100000;
double step;

int main() {
  long i;
  double x, pi, sum = 0.0;

  step = 1.0 / (double) num_steps;
  for (i = 0; i < num_steps; i++) {
    x = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
  }

  pi = step * sum;
  printf("Approximation␣of␣Pi:␣%.10f\n", pi);
  return 0;
}
```

Measure the execution time using `omp_get_wtime()`.

For the best reproducible results, measuring execution times should be done on the Fulda HPC cluster! This applies to all subsequent time measurements!

Task 1b: First Parallel Version

- Develop a parallel variant using only the following OpenMP constructs:
  - `#pragma omp parallel`
  - `omp_set_num_threads()`, `omp_get_thread_num()`, `omp_get_num_threads()`
- **Do not use** `#pragma omp parallel for`.

- The main challenge is to divide the loop iterations among threads manually.
  - Use the thread ID and total number of threads to compute the iteration range for each thread.
  - Each thread should accumulate its own partial sum in a local variable.
  - After the parallel region, combine all partial sums to obtain the final value of π.

Hint

There are two main strategies for distributing work among threads:

- **Cyclic distribution:** Each thread processes every p-th iteration.
- **Block decomposition:** Each thread processes a contiguous block of iterations.

Task 2 (`pi_par_critical.c`)

Develop additional parallel implementations of π and compare their performance.

- Extend your previous program by experimenting with different parallelization strategies.
- You may now use the directive:
  - `#pragma omp critical`
- Implement at least two different parallel versions of the π computation.
- Measure and compare their performance for different thread counts.

Evaluation

- Determine which implementation achieves the best performance.
- Create a table summarizing:
  - The variant name (e.g., *manual sum*, *critical*, . . .)
  - The number of threads used
  - The measured execution time (in seconds)
- Optionally, plot the results or calculate the achieved speedup.

Task 3 (`pi_par_loop.c`)

Develop additional parallel versions of the π program using `#pragma omp parallel for` and explore scheduling policies.

- Use `#pragma omp parallel for` to parallelize the main loop.
- Experiment with scheduling:
  - `schedule(static)`, `schedule(dynamic)`, `schedule(guided)`

  - Different `chunk` sizes
  - (Optional) `schedule(runtime)` via `OMP_SCHEDULE`
- Measure wall-clock time for multiple thread counts and settings.
- Extend your performance table from Task 2 with these variants.