```python
import time
import matplotlib.pyplot as plt
from projectpi.numbers import *
from projectpi.algorithms import *
from projectpi.pi_comparison import time_function, benchmark_methods,
plot_results, run_full_benchmark

# Create instances for testing
a_rational = Rational(1, 2)
b_rational = Rational(1, 3)

a_floatnum = FloatNumber(0.5)
b_floatnum = FloatNumber(0.3333333333333333)

a_fixed = FixedPrecisionNumber(0.5, q=5)
b_fixed = FixedPrecisionNumber(0.33333, q=5)

a_builtin = 0.5
b_builtin = 0.3333333333333333

# Now, define a function to perform multiple iterations and measure
time
def benchmark_operation(operation_func, *args, iterations=100000):
    def repeated_operation():
        for _ in range(iterations):
            operation_func(*args)
    _, exec_time = time_function(repeated_operation)
    avg_time = exec_time / iterations
    return avg_time

# Define all operations
operations = {
    'Addition': {
        'Rational': lambda: Rational.__add__(a_rational, b_rational),
        'FloatNumber': lambda: FloatNumber.__add__(a_floatnum,
b_floatnum),
        'FixedPrecisionNumber': lambda:
FixedPrecisionNumber.__add__(a_fixed, b_fixed),
        'Builtin float': lambda: float.__add__(a_builtin, b_builtin)
    },
    'Subtraction': {
        'Rational': lambda: Rational.__sub__(a_rational, b_rational),
        'FloatNumber': lambda: FloatNumber.__sub__(a_floatnum,
b_floatnum),
        'FixedPrecisionNumber': lambda:
FixedPrecisionNumber.__sub__(a_fixed, b_fixed),
        'Builtin float': lambda: float.__sub__(a_builtin, b_builtin)
    },
    'Multiplication': {
        'Rational': lambda: Rational.__mul__(a_rational, b_rational),
```

```
        'FloatNumber': lambda: FloatNumber.__mul__(a_floatnum,
b_floatnum),
        'FixedPrecisionNumber': lambda:
FixedPrecisionNumber.__mul__(a_fixed, b_fixed),
        'Builtin float': lambda: float.__mul__(a_builtin, b_builtin)
    },
    'Division': {
        'Rational': lambda: Rational.__truediv__(a_rational,
b_rational),
        'FloatNumber': lambda: FloatNumber.__truediv__(a_floatnum,
b_floatnum),
        'FixedPrecisionNumber': lambda:
FixedPrecisionNumber.__truediv__(a_fixed, b_fixed),
        'Builtin float': lambda: float.__truediv__(a_builtin,
b_builtin)
    },
    'Exponentiation': {
        'Rational': lambda: Rational.__pow__(a_rational, 2),
        'FloatNumber': lambda: FloatNumber.__pow__(a_floatnum, 2),
        'FixedPrecisionNumber': lambda:
FixedPrecisionNumber.__pow__(a_fixed, 2),
        'Builtin float': lambda: float.__pow__(a_builtin, 2)
    }
}

# Perform the benchmarking
def perform_benchmark():
    iterations = 100000  # Number of iterations per operation

    for op_name, op_funcs in operations.items():
        print(f"\n--- {op_name} ---")
        for class_name, func in op_funcs.items():
            avg_time = benchmark_operation(func,
iterations=iterations)
            print(f"{class_name}: {avg_time * 1e6:.3f} µs per
operation")

if __name__ == "__main__":
    perform_benchmark()
```

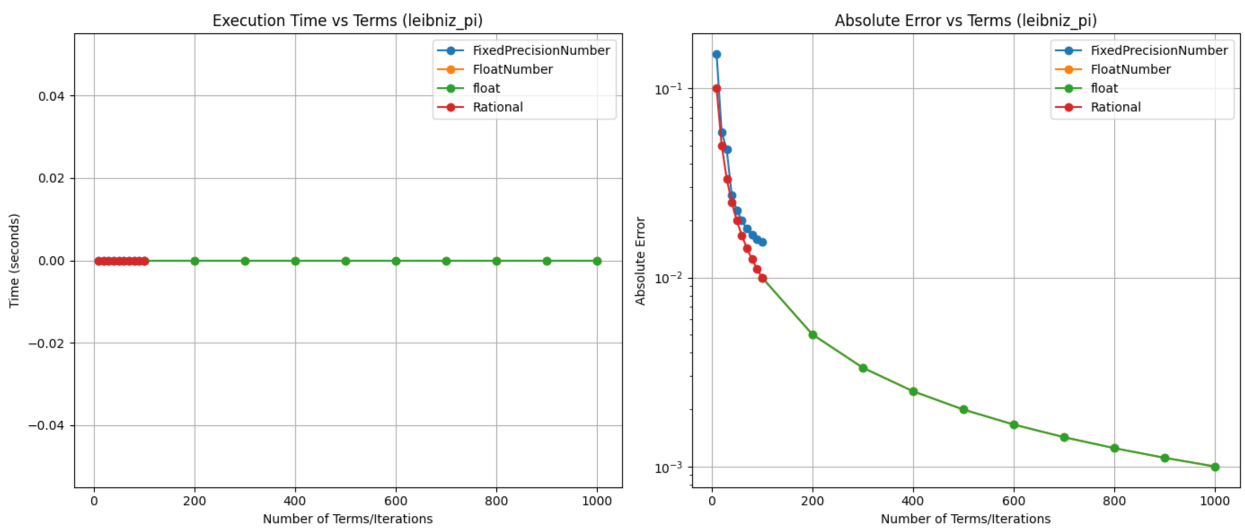
```
--- Addition ---
Rational: 0.388 µs per operation
FloatNumber: 1.875 µs per operation
FixedPrecisionNumber: 1.233 µs per operation
Builtin float: 0.135 µs per operation

--- Subtraction ---
Rational: 0.333 µs per operation
FloatNumber: 1.436 µs per operation
```

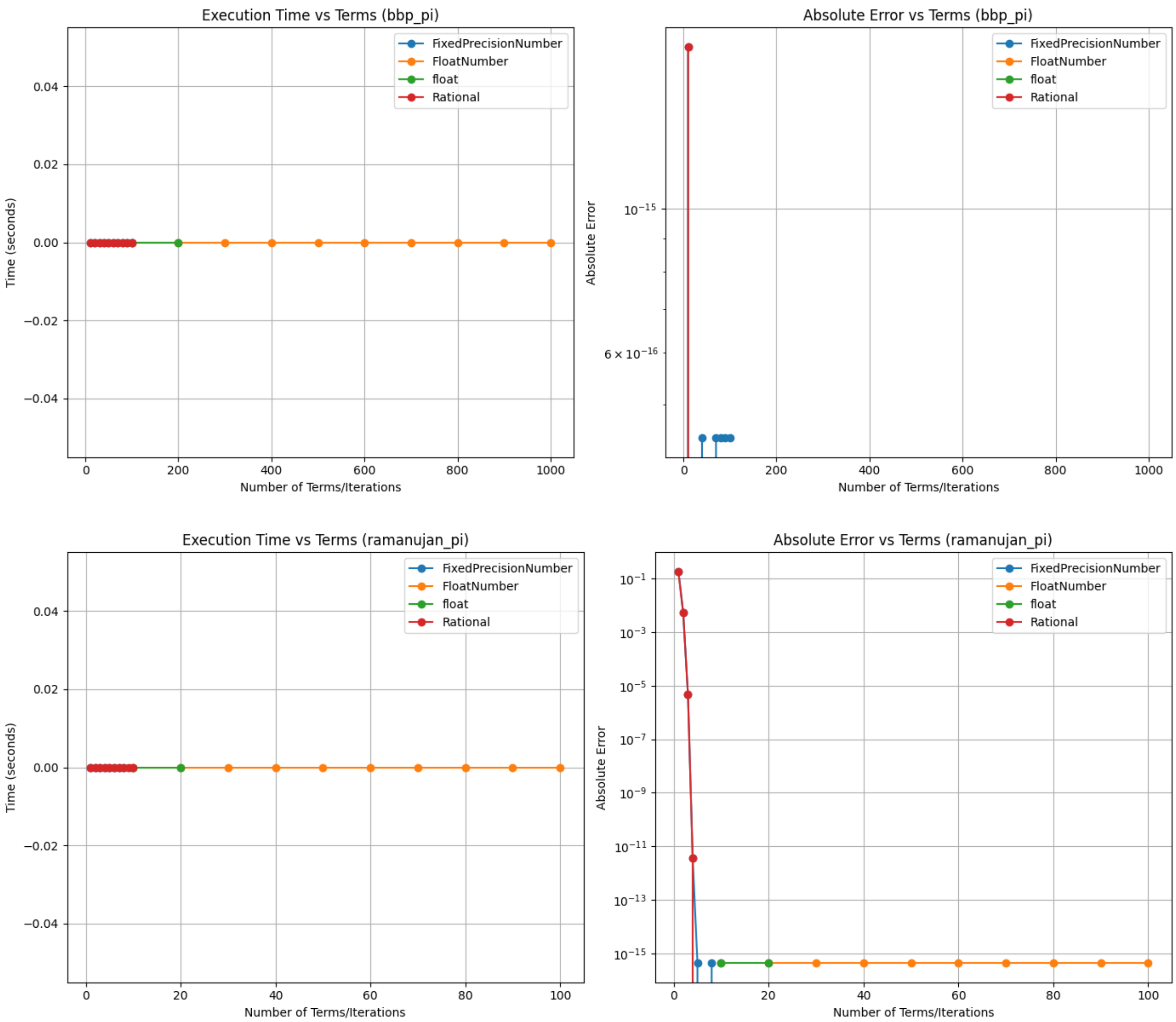
```
FixedPrecisionNumber: 1.247 µs per operation
Builtin float: 0.139 µs per operation

--- Multiplication ---
Rational: 0.307 µs per operation
FloatNumber: 1.337 µs per operation
FixedPrecisionNumber: 1.393 µs per operation
Builtin float: 0.126 µs per operation

--- Division ---
Rational: 0.314 µs per operation
FloatNumber: 2.785 µs per operation
FixedPrecisionNumber: 1.379 µs per operation
Builtin float: 0.136 µs per operation

--- Exponentiation ---
Rational: 0.335 µs per operation
FloatNumber: 1.906 µs per operation
FixedPrecisionNumber: 0.947 µs per operation
Builtin float: 0.150 µs per operation

if __name__ == "__main__":
    run_full_benchmark()


Benchmarking leibniz_pi with FixedPrecisionNumber...
  Terms: 10, Error: 0.151593
  Terms: 20, Error: 0.058593
  Terms: 30, Error: 0.047593
  Terms: 40, Error: 0.027193
  Terms: 50, Error: 0.022693
  Terms: 60, Error: 0.019993
  Terms: 70, Error: 0.018093
  Terms: 80, Error: 0.016893
  Terms: 90, Error: 0.015993
  Terms: 100, Error: 0.015393

Benchmarking leibniz_pi with FloatNumber...
  Terms: 100, Error: 0.010000
  Terms: 200, Error: 0.005000
  Terms: 300, Error: 0.003333
  Terms: 400, Error: 0.002500
  Terms: 500, Error: 0.002000
  Terms: 600, Error: 0.001667
  Terms: 700, Error: 0.001429
  Terms: 800, Error: 0.001250
  Terms: 900, Error: 0.001111
  Terms: 1000, Error: 0.001000

Benchmarking leibniz_pi with float...
```

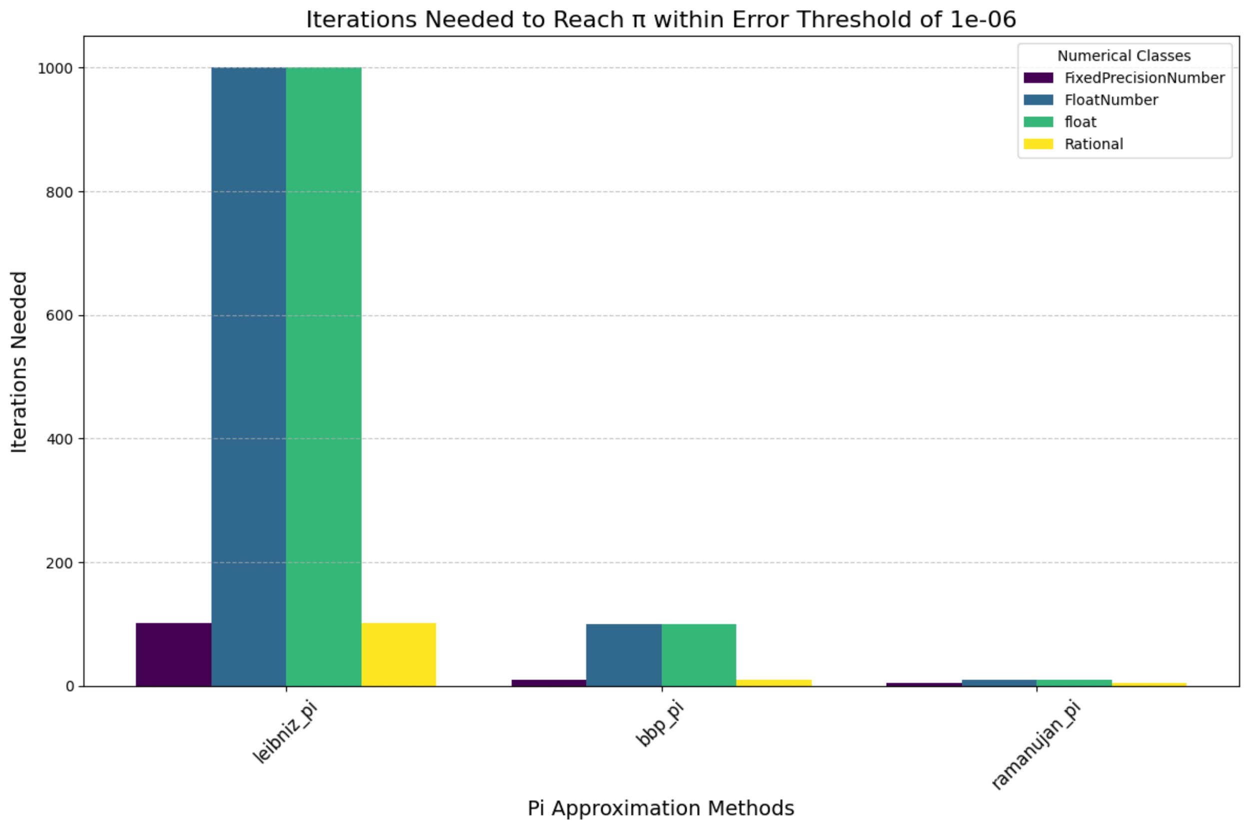
```
  Terms: 100, Error: 0.010000
  Terms: 200, Error: 0.005000
  Terms: 300, Error: 0.003333
  Terms: 400, Error: 0.002500
  Terms: 500, Error: 0.002000
  Terms: 600, Error: 0.001667
  Terms: 700, Error: 0.001429
  Terms: 800, Error: 0.001250
  Terms: 900, Error: 0.001111
  Terms: 1000, Error: 0.001000

Benchmarking leibniz_pi with Rational...
  Terms: 10, Error: 0.099753
  Terms: 20, Error: 0.049969
  Terms: 30, Error: 0.033324
  Terms: 40, Error: 0.024996
  Terms: 50, Error: 0.019998
  Terms: 60, Error: 0.016666
  Terms: 70, Error: 0.014285
  Terms: 80, Error: 0.012500
  Terms: 90, Error: 0.011111
  Terms: 100, Error: 0.010000

Benchmarking bbp_pi with FixedPrecisionNumber...
  Terms: 10, Error: 0.000000
  Terms: 20, Error: 0.000000
  Terms: 30, Error: 0.000000
  Terms: 40, Error: 0.000000
  Terms: 50, Error: 0.000000
  Terms: 60, Error: 0.000000
  Terms: 70, Error: 0.000000
  Terms: 80, Error: 0.000000
  Terms: 90, Error: 0.000000
  Terms: 100, Error: 0.000000

Benchmarking bbp_pi with FloatNumber...
  Terms: 100, Error: 0.000000
  Terms: 200, Error: 0.000000
  Terms: 300, Error: 0.000000
  Terms: 400, Error: 0.000000
  Terms: 500, Error: 0.000000
  Terms: 600, Error: 0.000000
  Terms: 700, Error: 0.000000
  Terms: 800, Error: 0.000000
  Terms: 900, Error: 0.000000
  Terms: 1000, Error: 0.000000

Benchmarking bbp_pi with float...
  Terms: 100, Error: 0.000000
  Terms: 200, Error: 0.000000
```

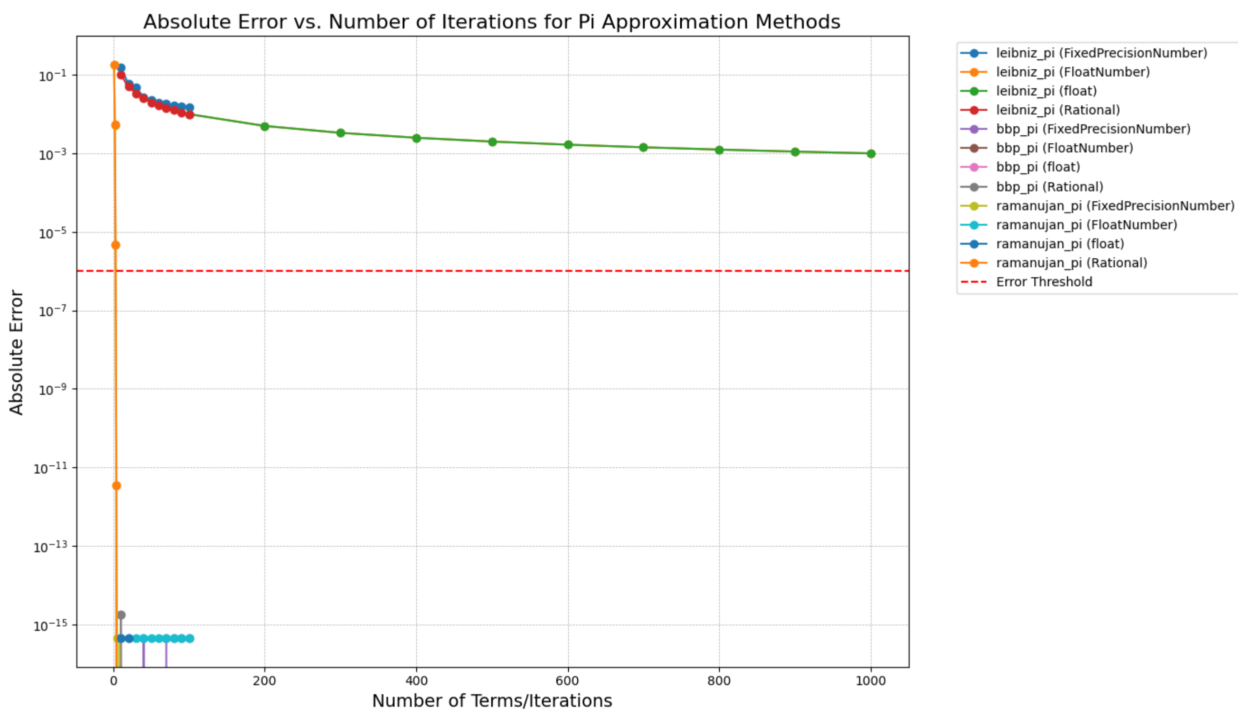
```
  OverflowError encountered at terms=300. Skipping...
  OverflowError encountered at terms=400. Skipping...
  OverflowError encountered at terms=500. Skipping...
  OverflowError encountered at terms=600. Skipping...
  OverflowError encountered at terms=700. Skipping...
  OverflowError encountered at terms=800. Skipping...
  OverflowError encountered at terms=900. Skipping...
  OverflowError encountered at terms=1000. Skipping...

Benchmarking bbp_pi with Rational...
  Terms: 10, Error: 0.000000
  Terms: 20, Error: 0.000000
  Terms: 30, Error: 0.000000
  Terms: 40, Error: 0.000000
  Terms: 50, Error: 0.000000
  Terms: 60, Error: 0.000000
  Terms: 70, Error: 0.000000
  Terms: 80, Error: 0.000000
  Terms: 90, Error: 0.000000
  Terms: 100, Error: 0.000000

Benchmarking ramanujan_pi with FixedPrecisionNumber...
  Terms: 1, Error: 0.181593
  Terms: 2, Error: 0.005293
  Terms: 3, Error: 0.000005
  Terms: 4, Error: 0.000000
  Terms: 5, Error: 0.000000
  Terms: 6, Error: 0.000000
  Terms: 7, Error: 0.000000
  Terms: 8, Error: 0.000000
  OverflowError encountered at terms=9. Skipping...
  OverflowError encountered at terms=10. Skipping...

Benchmarking ramanujan_pi with FloatNumber...
  Terms: 10, Error: 0.000000
  Terms: 20, Error: 0.000000
  Terms: 30, Error: 0.000000
  Terms: 40, Error: 0.000000
  Terms: 50, Error: 0.000000
  Terms: 60, Error: 0.000000
  Terms: 70, Error: 0.000000
  Terms: 80, Error: 0.000000
  Terms: 90, Error: 0.000000
  Terms: 100, Error: 0.000000

Benchmarking ramanujan_pi with float...
  Terms: 10, Error: 0.000000
  Terms: 20, Error: 0.000000
  OverflowError encountered at terms=30. Skipping...
  OverflowError encountered at terms=40. Skipping...
```

```
  OverflowError encountered at terms=50. Skipping...
  OverflowError encountered at terms=60. Skipping...
  OverflowError encountered at terms=70. Skipping...
  OverflowError encountered at terms=80. Skipping...
  OverflowError encountered at terms=90. Skipping...
  OverflowError encountered at terms=100. Skipping...

Benchmarking ramanujan_pi with Rational...
  Terms: 1, Error: 0.179671
  Terms: 2, Error: 0.005440
  Terms: 3, Error: 0.000005
  Terms: 4, Error: 0.000000
  Terms: 5, Error: 0.000000
  Terms: 6, Error: 0.000000
  Terms: 7, Error: 0.000000
  Terms: 8, Error: 0.000000
  Terms: 9, Error: 0.000000
  Terms: 10, Error: 0.000000
```




Execution Time vs Terms (bbp_pi)
Time (seconds)
Number of Terms/Iterations
FixedPrecisionNumber
FloatNumber
float
Rational
Absolute Error vs Terms (bbp_pi)
Absolute Error
Number of Terms/Iterations
FixedPrecisionNumber
FloatNumber
float
Rational
Execution Time vs Terms (ramanujan_pi)
Time (seconds)
Number of Terms/Iterations
FixedPrecisionNumber
FloatNumber
float
Rational
Absolute Error vs Terms (ramanujan_pi)
Absolute Error
Number of Terms/Iterations
FixedPrecisionNumber
FloatNumber
float
Rational


Iterations Needed to Reach π within Error Threshold of 1e-06
Numerical Classes
FixedPrecisionNumber
FloatNumber
float
Rational
Iterations Needed
0
200
400
600
800
1000
leibniz_pi
bbp_pi
ramanujan_pi
Pi Approximation Methods


Absolute Error vs. Number of Iterations for Pi Approximation Methods
leibniz_pi (FixedPrecisionNumber)
leibniz_pi (FloatNumber)
leibniz_pi (float)
leibniz_pi (Rational)
bbp_pi (FixedPrecisionNumber)
bbp_pi (FloatNumber)
bbp_pi (float)
bbp_pi (Rational)
ramanujan_pi (FixedPrecisionNumber)
ramanujan_pi (FloatNumber)
ramanujan_pi (float)
ramanujan_pi (Rational)
Error Threshold
Absolute Error
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
0
200
400
600
800
1000
Number of Terms/Iterations

Comparative Analysis of Pi Approximation Methods

Accurately approximating the mathematical constant π (pi) has been a longstanding endeavor in mathematics and computational science. Various algorithms have been developed, each leveraging different mathematical principles to estimate π with varying degrees of efficiency and accuracy. This report provides a concise overview of five prominent π approximation methods implemented in the provided Python code: **Leibniz Formula**, **Polygon Method**, **Monte Carlo Method**, **Bailey–Borwein–Plouffe (BBP) Formula**, and **Ramanujan's Series**. Additionally, it highlights their computational characteristics and suitability for different applications.

1. Leibniz Formula for π

Description:
The Leibniz Formula expresses π as an infinite alternating series:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots\right)$$

Implementation Highlights:

- **Method:** Summation of terms with alternating signs.
- **Convergence Rate:** Extremely slow; requires a large number of terms to achieve high precision.
- **Computational Efficiency:** Low efficiency due to the need for many iterations to reduce error.
- **Accuracy:** Limited by its slow convergence; practical for educational purposes but not suitable for high-precision requirements.

Use Case:
Best suited for introductory demonstrations of infinite series and alternating series convergence rather than precise computational applications.

2. Polygon Method (Liu Hui's Algorithm)

Description:
Inspired by Archimedes' approach, the Polygon Method approximates π by inscribing and circumscribing polygons around a circle and calculating their perimeters. As the number of polygon sides increases, the approximation of π improves.

Implementation Highlights:

- **Method:** Iteratively doubling the number of polygon sides and recalculating the side lengths using the square root function.
- **Convergence Rate:** Moderate; faster than the Leibniz Formula but slower compared to more advanced series.

- **Computational Efficiency:** Reasonable, but computationally intensive for a high number of iterations due to repeated square root calculations.

- **Accuracy:** Improves steadily with each iteration; suitable for intermediate precision needs.

Use Case:
Effective for historical demonstrations of geometric approximation methods and for applications requiring moderate precision without the need for advanced mathematical computations.

3. Monte Carlo Method

Description:
The Monte Carlo Method estimates π by randomly sampling points within a unit square and determining the ratio that falls inside the inscribed quarter-circle. This probabilistic approach leverages statistical principles to approximate π.

Implementation Highlights:

- **Method:** Randomly generates points and calculates the proportion within a quarter-circle to estimate π.

- **Convergence Rate:** Relatively slow; accuracy improves with the square root of the number of samples.

- **Computational Efficiency:** Dependent on the number of samples; can be parallelized but generally less efficient for high precision.

- **Accuracy:** Suitable for low to moderate precision; statistical fluctuations can affect reliability at lower sample sizes.

Use Case:
Ideal for demonstrating probabilistic methods and for applications where simplicity is favored over precision, such as educational simulations and preliminary computational experiments.

4. Bailey–Borwein–Plouffe (BBP) Formula

Description:
The BBP Formula provides a rapidly converging series for π and uniquely allows the extraction of hexadecimal or binary digits of π without computing preceding digits. The formula is as follows:

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

Implementation Highlights:

- **Method:** Summation of a series with rapidly decreasing terms.

- **Convergence Rate:** High; achieves significant precision with fewer iterations compared to traditional series.

- **Computational Efficiency:** Highly efficient for high-precision calculations; suitable for digit extraction.

- **Accuracy:** Excellent; capable of producing millions of accurate digits of π with appropriate computational resources.

Use Case:
Preferred for high-precision computations, cryptographic applications, and scenarios requiring the extraction of specific digits of π without calculating the entire sequence.

5. Ramanujan's Series for π

Description:
Srinivasa Ramanujan developed several rapidly converging series for π, leveraging deep insights from number theory and infinite series. One such series is:

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\sum_{k=0}^{\infty}\frac{(4k)!(1103+26390k)}{(k!)^4 396^{4k}}$$

Implementation Highlights:

- **Method:** Summation of a highly optimized infinite series with factorial and exponential terms.

- **Convergence Rate:** Extremely fast; each additional term adds approximately eight correct digits of π.

- **Computational Efficiency:** Highly efficient for achieving ultra-high precision with minimal iterations; however, requires handling of large factorials and exponents.

- **Accuracy:** Exceptional; capable of generating millions of accurate digits of π swiftly.

Use Case:
Ideal for applications demanding the highest precision, such as scientific simulations, high-precision engineering calculations, and mathematical research requiring extensive digits of π.

Comparative Summary

Method	Convergence Rate	Iterations for High Precision	Computational Efficiency	Accuracy	Ideal Use Case
Leibniz Formula	Very Slow	~10^6 terms for millisecond precision	Low	Low	Educational demonstrations
Polygon Method	Moderate	~100 iterations for decent precision	Moderate	Moderate	Historical demonstrations, intermediate precision

Method	Convergence Rate	Iterations for High Precision	Computational Efficiency	Accuracy	Ideal Use Case
Monte Carlo Method	Slow	~10^4 samples for moderate precision	Variable	Moderate	Educational simulations, probabilistic studies
BBP Formula	Fast	~100 iterations for high precision	High	High	High-precision computations, digit extraction
Ramanujan's Series	Extremely Fast	~5 iterations for millions of digits	Very High	Very High	Ultra-high precision applications, mathematical research

Conclusion:
Selecting the appropriate π approximation method depends largely on the required precision, computational resources, and specific application needs.