The University Of Sheffield.

Department
Of
Mechanical
Engineering

MEng Mechanical Engineering

Identifying Dynamic Systems with Probabilistic Numerics

Harvey WALTON

December 2023

Supervisor: Dr Tim Rogers

Report submitted to the University of Sheffield in partial fulfilment of the requirements for the degree of Master of Engineering

Contents

1 Background and Understanding of the Problem

Problems in engineering are usually described using a framework of continuous mathematical functions. This means that their output has no jumps or gaps in values, and they can be evaluated using input values that are on a sliding scale of a continuous input domain. For example, time in the real world is on a continuous sliding scale, which means that any interval in time can always be subdivided into a smaller interval. Continuous functions in turn can often be manipulated analytically using mathematics. For example, if the velocity of a particle is given as a continuous function of time, a continuous acceleration function can be derived from the velocity analytically through a mathematical process called differentiation.

However, there are many situations where this analytical approach is unsuitable. The first reason for this is that the analytical approach may be too complex for anyone to have solved. A good example of this is the Gaussian (normal) distribution. This definite integral of this is often needed to be found in order to find the cumulative distribution function. This process can be visualised as finding the area underneath th

However, to collect data for real world events, it must be sampled or else the amount of information needed to represent the event, as well as the amount of computing power to perform analysis on the event, would tend to infinity. Similarly, many problems in the real world are too complex to be solved analytically and so numerical methods are used to approximate the solution to these problems. However, what if there was a better approach? What if instead of resorting to numerical methods to solve the problem discretely, the original continous data could be reconstructed in “closed form“.

However, these methods are not perfect, and introduce error into the solutions. This error can be quantified, and is often done so using error bounds

However, instead, what if the error was quantified using a probability distribution

Instead, what if we use uncertainty define a pdf of result using (mean and standard deviation) in this case, it may be easier to quantify concepts like risk and improve automation of decision-making within engineering

1.1

2 Aims and Objectives

Your text goes here.

3 Work completed to date

Your text goes here. [1]

4 Plan for future work

Your text goes here.

A The use of generative AI (ChatGPT)

OpenAI's ChatGPT interface, powered by the GPT-4 large language model, was used to create some code.

A.1 _inverse_triangluar_matrix function

This method is used to calculate the inverse of R in my method that computes the NLML in my FITC sparse GP approximation. It was generated using ChatGPT using the prompt: "Write a Python function named _inverse_upper_triangular that calculates the inverse of an upper triangular matrix in a faster and more numerically stable way than the standard np.linalg.inv function". The output is provided in Figure 1.:

ChatGPT

Based on your prompt, here's the Python function **'_inverse_upper_triangular'** that meets the specified requirements:

```python
import numpy as np

def _inverse_upper_triangular(matrix):
    # Convert the input matrix to a NumPy array for easier manipulation
    A = np.array(matrix, dtype=float)

    # Check if the matrix is square
    rows, cols = A.shape
    if rows != cols:
        raise ValueError("Input matrix is not square")

    # Check if the matrix is upper triangular
    if not np.allclose(np.tril(A, k=-1), 0):
        raise ValueError("Input matrix is not upper triangular")

    # Check for zeros on the main diagonal
    if any(np.isclose(np.diag(A), 0)):
        raise ValueError("Matrix is singular and has no inverse")

    # Create an identity matrix of the same size
    I = np.identity(rows)

    # Compute the inverse
    for i in reversed(range(rows)):
        I[i, i] = 1 / A[i, i]
        for j in reversed(range(i + 1, rows)):
            I[i, j] = -sum(A[i, k] * I[k, j] for k in range(i + 1, j + 1)) / A[i, i]

    return I

# Example usage
matrix = [[1, 2, 3], [0, 4, 5], [0, 0, 6]]
inverse_matrix = _inverse_upper_triangular(matrix)
print(inverse_matrix)
```

Figure 1: ChatGPT output providing a function used to inverse an upper triangular matrix in a faster and more numerically stable way

After this was generated, it was modified slightly to make it work in my FITC class. Firstly, it was made a method instead of a function by adding “self“ to the inputs. Secondly, the output was transformed into a np.array type which is expected for the subsequent code. The resulting method is provided below:

```python
def _inverse_upper_triangular(self, matrix):
    # Convert the input matrix to a NumPy array for easier manipulation
    A = np.array(matrix, dtype=float)

    # Check if the matrix is square
    rows, cols = A.shape
    if rows != cols:
        raise ValueError("Input matrix is not square")

    # Check if the matrix is upper triangular
    if not np.allclose(np.tril(A, k=-1), 0):
        raise ValueError("Input matrix is not upper triangular")

    # Check for zeros on the main diagonal
    if any(np.isclose(np.diag(A), 0)):
        raise ValueError(
            "Matrix is singular and has no inverse")  # Changed from return None for consistency

    # Create an identity matrix of the same size
    I = np.identity(rows)

    # Compute the inverse
    for i in reversed(range(rows)):
        I[i, i] = 1 / A[i, i]
        for j in reversed(range(i + 1, rows)):
            I[i, j] = -sum(A[i, k] * I[k, j] for k in range(i + 1, j + 1)) / A[i, i]

    return np.array(I.tolist())

```

This was proven to work by writing a unit test:

```python
def test__inverse_upper_triangular(self):
    obj = GP_NLL_FITC(1, 2, 3, 4, 5, 6)

    # Upper triangular matrix
    matrix = np.array([
        [2, 3, 1],
        [0, 4, 5],
        [0, 0, 7]
    ])
```

```
    start_time = timer.time()

    result = obj._inverse_upper_triangular(matrix)

    end_time = timer.time()
    elapsed_time = end_time - start_time
    print(f"The _inverse_upper_triangular func ran in {elapsed_time} seconds")

    start_time = timer.time()

    correct = np.linalg.inv(matrix)

    end_time = timer.time()
    elapsed_time = end_time - start_time
    print(f"The np.linalg.inv func ran in {elapsed_time} seconds")

    debug_print(f"result = {result}")
    debug_print(f"correct = {correct}")
    debug_print(f"difference = {result - correct}")

    assert np.allclose(result, correct, atol=1E-5, rtol=1E-5)

    # Another upper triangular matrix with different values
    matrix2 = np.array([
        [-1.9883, 2.0, 3.0],
        [0, -9.4199, 4.0],
        [0, 0, 1.7969]
    ])
    start_time = timer.time()

    result2 = obj._inverse_upper_triangular(matrix2)

    end_time = timer.time()
    elapsed_time = end_time - start_time
    print(f"The _inverse_upper_triangular func ran in {elapsed_time} seconds")

    start_time = timer.time()

    correct2 = np.linalg.inv(matrix2)

    end_time = timer.time()
    elapsed_time = end_time - start_time
    print(f"The np.linalg.inv func ran in {elapsed_time} seconds")
```

```
        debug_print(f"result2 = {result2}")
        debug_print(f"correct2 = {correct2}")
        debug_print(f"difference2 = {result2 - correct2}")

        assert np.allclose(result2, correct2, atol=1E-5, rtol=1E-5)

```

This compared the result calculated by the method with an expected result calculated using the standard `np.linalg.inv` function for the two different input matrices. If they were the same within both a relative and absolute tolerance of $1E-5$.

The unit test passed, showing that the function inverses the matrix correctly, however, it wasn't necessarily faster than the standard `np.linalg.inv` function. The inverse of the first matrix was calculated in 0.000246 and 0.001368 seconds using `_inverse_upper_triangular` and `np.linalg.inv` methods respectively, whereas the inverse of the second matrix was calculated in 0.000312 and 0.00005 seconds respectively. This indicates that the fastest method depends on the input matrix, which is likely true for larger matrices also since both functions are of the $O(n^3)$ for an $n \times n$ matrix. However, the `_inverse_upper_triangular` function will be more numerically stable and memory efficient because it is taking adavantages of the fact that the input matrix is upper triangular which allows the inverse to be found by back substitution instead of it being computed naively.

A.2 Figures for the report

A.2.1 Cumulative Distribution Function

Put prompt and output

References

[1] J. Quinonero-Candela and C. Rasmussen. "A Unifying View of Sparse Approximate Gaussian Process Regression". In: *Journal of Machine Learning Research* 6 (2005).