APPLICATIONS OF GREEN'S THEOREM FOR NUMERICAL QUADRATURE

May 22, 2023

Abstract

Certain integrals of two variables over a planar region are divergent or may take significant wall time when computed numerically. This paper presents the usage of Green's theorem to transform such integrals over a plane to a line integral over its boundary. This can reduce the order of a numerical integration from $O(N^2)$ to $O(N)$ for a class of functions defined on regions bounded by C^1 curves. We describe a general method of applying Green's theorem to such integrals, as well as the improved convergence and wall times as a result of this application. Additionally, we introduce a new Python package, called Vectorcalc, which aids in computing numerical line integrals. Experimental results demonstrate the effectiveness of the proposed method compared to traditional double quadrature methods, showcasing its ability to converge efficiently for highly variable integrals. The application of Green's theorem offers a practical solution for numerical quadrature in cases where traditional methods fail to converge or are computationally intensive.

Key words and phrases: Green's theorem; Numerical integration; Improved convergence; Reduced wall times

1 Introduction

Highly variable functions of two variables may be difficult to integrate over a 2-dimensional plane as the domain of integration grows by quadratic order. This can lead to traditional methods of quadrature failing to converge. Other authors have previously described the application of Green's theorem in simplifying the computation of geometric moments and scattering problems, but have not described an algorithm for reducing the 2-dimensional integrals to 1-dimensional integrals in the general sense (Yang and Albregtsen, 1996; Gordon and Billow, 2002; Li and Shen, 2012).

The general algorithm for implementing Green's theorem can be applied to integrals of the form

$$\int_c^d \int_a^b f(x,y)\,dx\,dy \tag{1}$$

for $f : \mathbb{R}^2 \to \mathbb{R}$. The application requires that the domain of integration, D, is bounded by four C^1 curves and the boundary of D is easily parameterized. Furthermore, we find that the practicality of transforming a 2-dimensional integral to a line integral is most useful for functions f such that

$$f(x,y) = g(x,y) - h(x,y) \tag{2}$$

where $g, h : \mathbb{R}^2 \to \mathbb{R}$, and g and h are trivially integrable with respect to x and y respectively. In the remainder of this paper, we:

- Describe the transformation done by Green's theorem
- Introduce a new Python package to aid in computing numerical line integrals
- Illustrate the resulting improvements over traditional double quadrature methods.

2 Using Green's Theorem

Green's theorem states

$$\int\int_D \frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y} = \oint_{\partial D} (Pdx + Qdy) \tag{3}$$

for functions $Q, P : \mathbb{R}^2 \to \mathbb{R}$ defined on a region D bounded by four C^1 curves. It reduces the 2-dimensional integral over D to a line integral over its boundary. This has practical applications in calculating line integrals analytically, but as we find in this paper, it can also be used to improve the convergence of highly variable functions.

The algorithm is described for $\emptyset \neq D \subseteq \mathbb{R}^N, f : D \to \mathbb{R}^M$:

1. Write f as a difference of two functions, $f = g - h$, where g, h are trivially integrable with respect to x and y, respectively.
2. Analytically integrate g with respect to x and h with respect to y
3. Numerically integrate the sum of these integrals over the boundary of D:
$$\oint_{\partial D} \left(\int -h \, dy \right) dx + \left(\int g \, dx \right) dy \, . \tag{4}$$

To illustrate this, we begin with an example, by finding the numerical integral of the equation:

$$\int_0^{10} \int_0^{10} \cos(e^x) + \sin(e^y) \, dx \, dy \tag{5}$$

which is over the square in $\mathbb{R}^2$ with sidelength 10. From inspection, it is clear that the function is highly variable (In fact, traditional quadrature methods will not converge. This will be explored further in section 4.), the domain of integration is bounded by C^1 curves, and the function can be written as a difference of two trivially integrable functions in x and y, thus making it a prime candidate for this method. First, we rewrite the equation:

$$f(x, y) = \cos(e^x) + \sin(e^y) = \sin(e^y) - (-\cos(e^x)). \tag{6}$$

Then we integrate each summand in (6),

$$\int \sin(e^y) \, dx = x \sin(e^y), \int -\cos(e^x) \, dy = -y \cos(e^x).$$

Finally, we can numerically integrate over the perimeter of the square,

$$\int_{\partial D} -y \cos(e^x) \, dx + x \sin(e^y) \, dy \tag{7}$$

to find the value of equation (5).

3 Introducing Vectorcalc (Python Package)

To calculate line integrals using numerical methods, we developed a Python package that interprets line parameterizations and conducts a Riemann integral with the aid of a midpoint technique. Specifically, given a curve

$\gamma : [a, b] \to \mathbb{R}^N$ and a function $f : \{\gamma\} \to \mathbb{R}^N$, the package generates a partition $a = t_0 < t_1 < \cdots < t_n = b$ and returns

$$\sum_{i=1}^{n} f(\gamma(t_{i+1} + \frac{t_i - t_{i-1}}{2})) \cdot (\gamma(t_i) - \gamma(t_{i-1})), \tag{8}$$

where n is set to 100 by default.

Returning to our example in equation (7), we begin by defining our functions in Python:

Code Block 1: Defining the Integrand of Equation 7 in Python

```
import vectorcalc as vc
from math import *

def g(x, y):
    return x*sin(exp(y))
def h(x, y):
    return -y*cos(exp(x))
```

By using Vectorcalc's vector_integrate_square() function, we can calculate the numerical line integral over a square with one point at $(0, 0)$ and another point at $(10, 10)$:

Code Block 2: Numerically Integrating Equation 7

```
vc.vector_integrate_square([h, g],
                           [0, 0], [10, 10],
                           neval=100000)
```

where *neval* is equivalent to n in (8). Running this code returns:

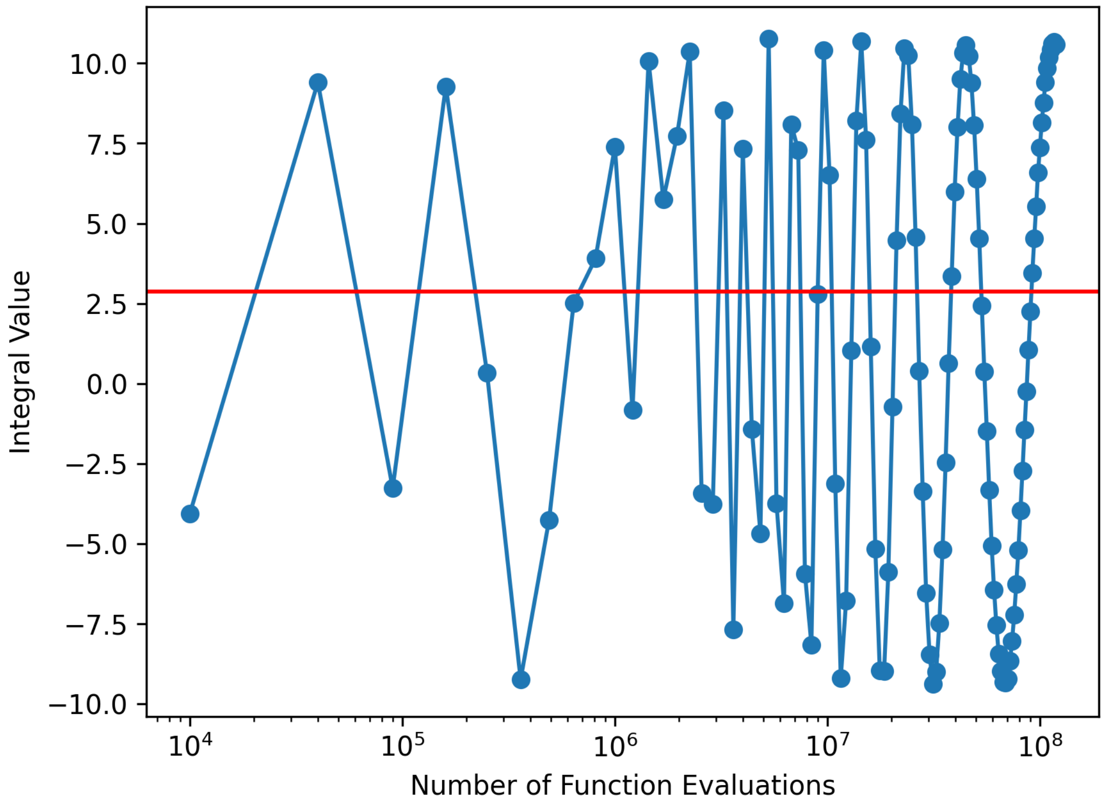
```
2.8731137403155196
```

This is compared to the analytic result of 2.8731098618335261.

4 Comparison to Traditional Methods

As mentioned earlier, this method not only reduces the dimensionality of the problem, but it can also converge to the theoretical value where traditional methods may fail. The results of this can be seen in the figures below.

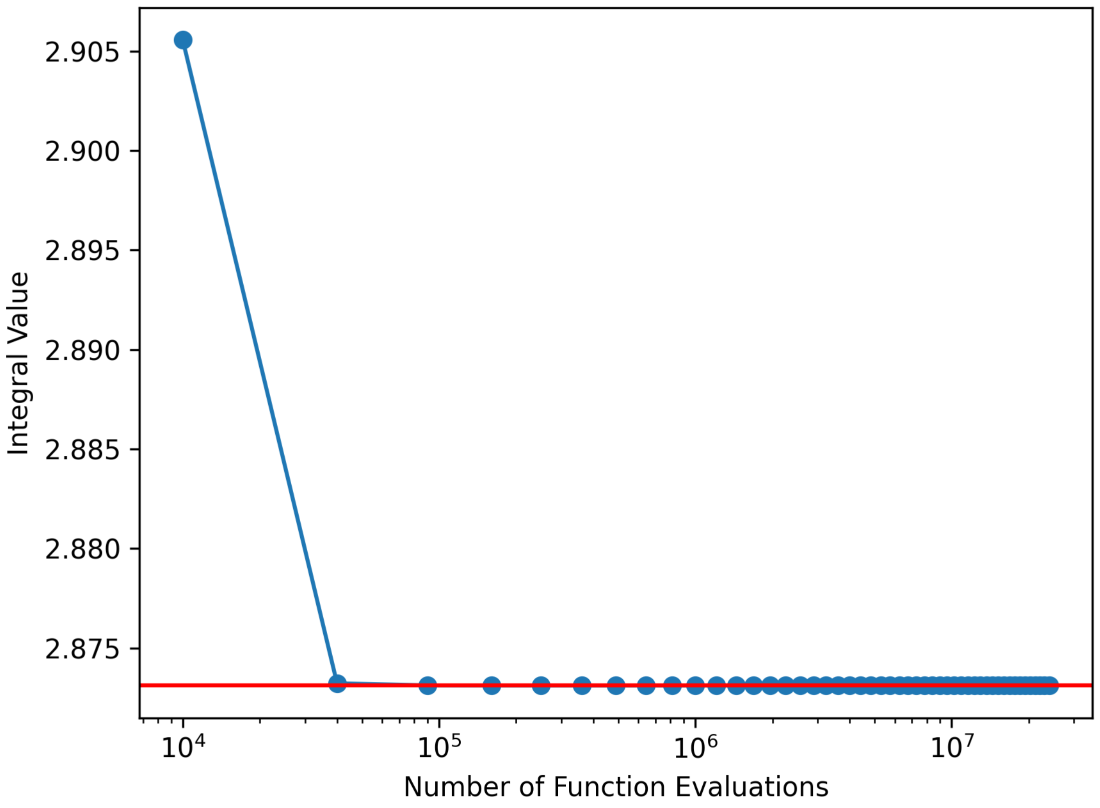
Figure 1: **Double Midpoint Integration of Eq. 5.** A double midpoint integration of our example. The theoretical value is depicted as a red line here. Note the log scale of the x-axis. The double midpoint integration code can be found in the Appendix.



Clearly, the double midpoint integration method struggles to converge to the theoretical value due to the highly variable nature of the function. As we increase the number of function evaluations, the result of the integral oscillates around the theoretical value but does not converge. This highlights the limitations of traditional methods in dealing with complicated functions.

On the other hand, the application of Green's Theorem to reduce the dimensionality of our integration region allows the integral to converge to the theoretical value much more efficiently. This can be seen in the figure below.

Figure 2: **Green's Theorem Integration of Eq. 7.** The application of Green's Theorem to reduce the dimensionality of our integration region also allows the integral to converge to the theoretical value (depicted as a red line). Note, the same number of function evaluations are used from Figure 1.



As shown in the figure, the application of Green's Theorem reduces the complexity of the problem and allows the integral to converge much more efficiently to the theoretical value. This demonstrates the effectiveness of the proposed method compared to traditional methods in dealing with highly variable integrals.

5 Conclusion

We have demonstrated that for highly-variable integrals over regions bounded by C^1 curves, the application of Green's Theorem is able to reduce the dimensionality of the problem and converge faster than traditional methods. It can also be shown that compared to more modern numerical integration methods, namely Scipy's `dblquad`, the application of Green's Theorem still offers superior speed and convergence. However, as `dblquad` takes more time for each function evaluation, the convergence can not be as easily compared. Furthermore, we have only demonstrated the use of this method for a function where the analytic value was known, but this method could also be applied to non-analytically integrable functions that may fail converge numerically.

6 References

Gordon, W.B. and Bilow, H.J. (2002) 'Reduction of surface integrals to contour integrals', IEEE Transactions on Antennas and Propagation, 50(3), pp. 308–311. doi:10.1109/8.999621.

Li, B.-C. and Shen, J. (1991) 'Fast computation of moment invariants', Pattern Recognition, 24(8), pp. 807–813. doi:10.1016/0031-3203(91)90048-a.

Yang, L. and Albregtsen, F. (1996) 'Fast and exact computation of Cartesian geometric moments using discrete Green's theorem', Pattern Recognition, 29(7), pp. 1061–1073. doi:10.1016/0031-3203(95)00147-6.

7 Appendix

7.1 Figures

Figure 1: **Double Midpoint Integration of Eq. 5.** A double midpoint integration of our example. The theoretical value is depicted as a red line here. Note the log scale of the x-axis.

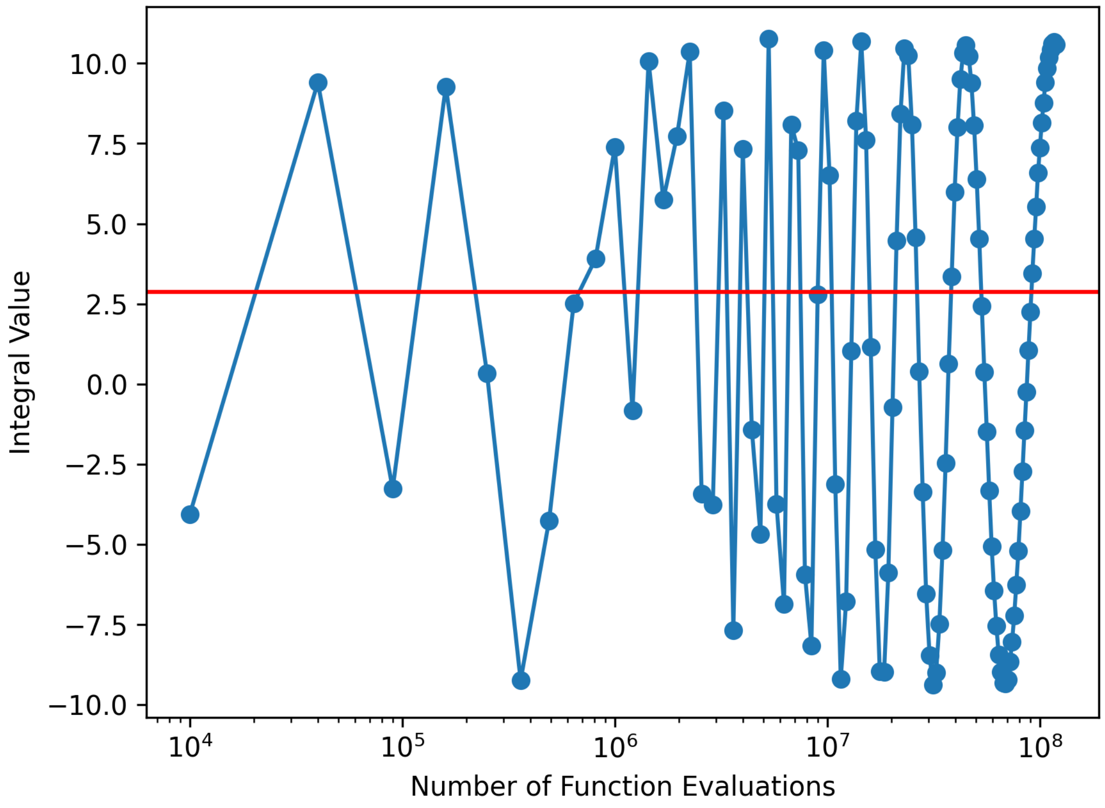
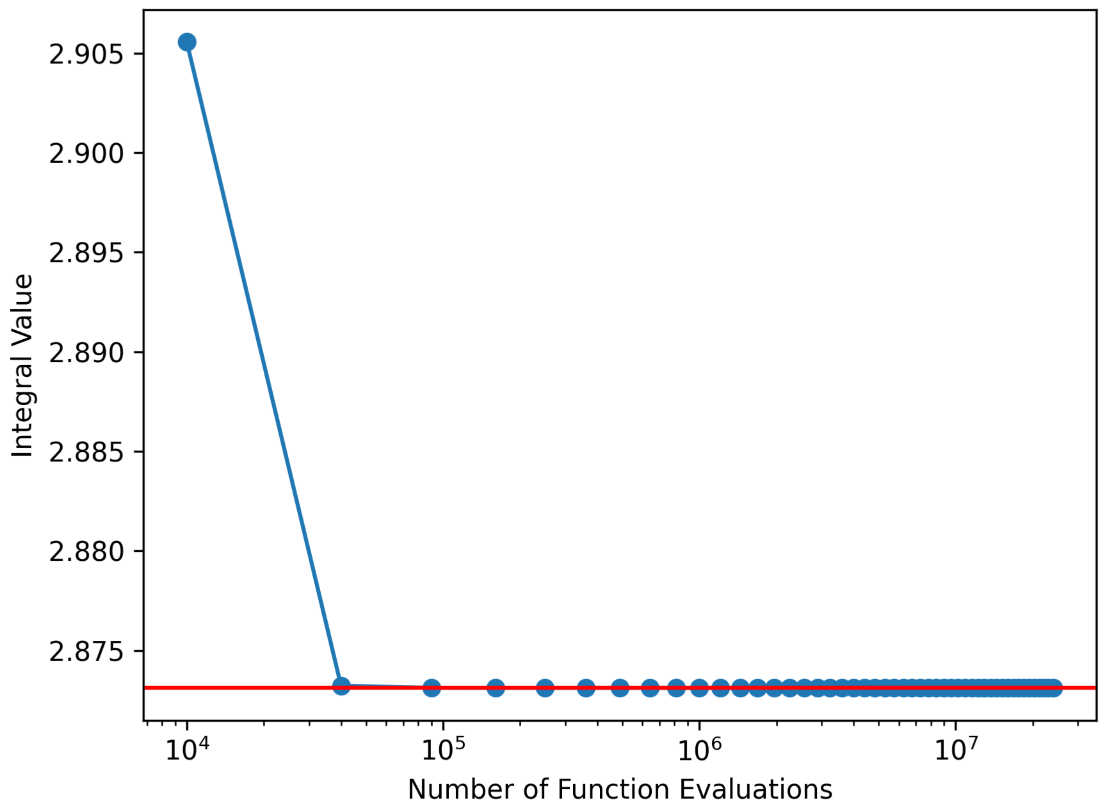


Figure 2: **Green's Theorem Integration of Eq. 7.** The application of Green's Theorem to reduce the dimensionality of our integration region also allows the integral to converge to the theoretical value (depicted as a red line). Note, the same number of function evaluations are used from Figure 1.



7.2 Code

Code Block 1: Defining the Integrand of Equation 7 in Python

```
import vectorcalc as vc
from math import *

def g(x, y):
    return x*sin(exp(y))
def h(x, y):
    return -y*cos(exp(x))
```

Code Block 2: Numerically Integrating Equation 7

```
vc.vector_integrate_square([h, g],
                           [0, 0], [10, 10],
                           neval=100000)
```

Code Block 3: Double Midpoint Method Used in Figure 1.

```
def dbl_midpoint(f, a, b, c, d, n):
    h = float(b-a)/n
    h_two = float(d-c)/n
    result = 0
    for i in range(n):
        for j in range(n):
            result += f((a + h/2.0) + i*h, (c + h/2.0) + j*h)
        result *= h_two
    result += h

    return result

def func(x, y):
    return cos(exp(x)) + sin(exp(y))
```