The Role of Improved Computational Speed and Parallel Processing in the Use of Numerical Integration Algorithms

To what extent are library algorithms for numerical integration faster or more accurate than directly implemented algorithms given the modern role of parallel processing and increased computational speed?

Irene Crowell

Session: May 2017

Session Number:

Word Count: 3826

Abstract

This essay examines the question of "To what extent are library algorithms for numerical integration faster or more accurate than directly implemented algorithms given the modern role of parallel processing and increased computational speed?" The focus of the investigation is on a comparison between the many existing numerical integration algorithms that are both widely available in scientific programming libraries and those that are less widely used, but simple to implement. In carrying out this comparison, special emphasis will be placed on how processing speed affects the practicality and usefulness of the different methods, as well as how that speed can be improved by the use of parallelization.

Following this investigation, it was found that the increased processing speed of modern computers results in a minimal difference in calculation time for the best algorithms, allowing for the use of a precise, general purpose algorithm in the form of the widely used Adaptive Gauss-Kronrod Quadrature. Parallelization is deemed to currently be largely insignificant to the speed of such algorithms, but its potential future role in regards to numerical integration is discussed and emphasized.

[Words: 181]

Table of Contents

Introduction

Numerical integration, or quadrature, has been a principal tool of mathematicians for centuries, even before the advent of calculus and the formalization of the definition of an integral. With applications in mechanical engineering, vision processing, and computer graphics[1], numerical integration is useful because of the large range of functions whose symbolic integrals are highly complex, or do not exist. The technique has become truly useful in recent history due to the rise of computers able to perform the required amount of accurate calculations quickly. To be effective, the chosen algorithm must be applicable to both those "badly behaved" functions that are impractical for symbolic calculation, as well as for all functions that cannot be symbolically integrated in a traditional programming language. Both tasks are necessary for the algorithm to be applied in the midst of a larger program that will be confronted with unknown functions.

Many attempts exist to produce such general purpose algorithms. The QUADPACK set of algorithms for FORTRAN has existed since 1983[2], and been re-implemented in variety of languages, including C through the GNU Scientific Library[3]. In contrast, simpler algorithms based on polynomial function approximations can be easily and directly implemented in any language. Little research exists on whether the library functions produce higher accuracy or greater speed, and if any one numerical integration algorithm can be applied universally. Such

[1] Solomon, Justin. "Chapter 14: Integration and Differentiation." Numerical Algorithms: Methods for Computer Vision, Machine Learning, and Graphics. N.p.: n.p., n.d. 277-302. Print.
[2] Piessens, R. Quadpack: A Subroutine Package for Automatic Integration. Berlin: Springer-Verlag, 1983. Print.
[3] "GNU Scientific Library -- Reference." GNU Scientific Library. Free Software Foundation, Inc., 31 Aug. 2016. Web. 16 Sept. 2016. <https://www.gnu.org/software/gsl/manual/>.

concerns may also be affected by the advent of parallel processing, which can be better applied by modifying simple algorithms as opposed to implemented library algorithms, and the increased speed of modern processors, which may lessen the importance of calculation speed in algorithm selection. Thus, this paper seeks to answer the question "To what extent are library algorithms for numerical integration faster or more accurate than directly implemented algorithms given the modern role of parallel processing and increased computational speed?"

Methodology

In order to attempt to answer the stated research question, it is necessary to evaluate numerous numerical integration algorithms and compare their effectiveness. C++ was chosen as the language of implementation due to its ease of optimization when compared to languages like Java, as well as my familiarity with its use. For comparison, several classes of algorithms were implemented in contrast with library algorithms, including "brute force" approaches, adaptive algorithms, and parallelized algorithms. When evaluating results, accuracy was given weight over speed[4], as small differences in time are generally forgivable, while inaccurate results can be extremely harmful to programs as a whole. Each algorithm was tested against a large range of functions[5], from simple polynomials and trigonometric functions, to complex functions with singularities or "bad behavior" that makes them difficult to evaluate. Also given weight was the presence of error estimates for several algorithms. Additionally, several algorithms offer error estimates that may make them more useful.

[4] Appendix: Tables 2-59
[5] Appendix: Table 1

"Brute Force" Algorithms

Algorithms of this type calculate the integral by evaluating the function at many points and approximating it as a simply integrated polynomial. The Newton-Cotes Rules are not readily available in programming libraries, while Gauss-Legendre and Gauss-Kronrod rules are.

Midpoint Rule and Newton-Cotes Rules

The simplest algorithm is the *Midpoint Rule*, which approximates the area as a series of many extremely thin rectangles, becoming more accurate as the number of rectangles increases and their width decreases. Other algorithms, known as *Newton-Cotes Rules*, approximate the function with polynomials of increasing orders, from linear trapezoids through quartics[6]. Comparing these algorithms involves examining the tradeoffs between lower order rules which are theoretically less accurate, and higher order rules, and whether this gap can be bridged by increasing the number of function evaluations, which can create slower processing times.

To understand the effects of this increased processing, data was examined from less and more processor heavy variations of the rules. In this instance, where there is no ability to specify an error tolerance, 10^2 subdivisions was taken as the less accurate measure, and 10^5 as the more accurate[7], where a subdivision is a section of the function to be approximated by a polynomial. The difference in time for calculation was as expected -- calculating 10^5 subdivisions yielded times on the order of 10^{-2} seconds, while 10^2 subdivisions yielded 10^{-5} seconds for nearly all algorithms of this class. There was slight variation, with the higher order Newton-Cotes Rules,

[6] Atkinson, Kendall E. "Numerical Integration." An Introduction to Numerical Analysis. New York: Wiley, 1978. 215-28. Print.

[7] Appendix: Tables 2-11

which evaluated the function several times on each subdivision, slightly slower, around 2 to 4 times than the lower order rules. It is important to note, however, that despite the large difference in magnitude, times for both accuracy goals were relatively small. Thus, faster modern processors are able to reduce the differences in speed between algorithms, and even between accuracies. Even for the more accurate calculations, from the perspective of the user, 100 different functions could be integrated before even a second's delay is noted.

In terms of accuracy, two different percent errors were considered -- the first being the average of all functions that were successfully integrated, and the second adjusted to ignore outliers of above 20%. Between rules, the unadjusted error generally decreased as the rule order increased for the higher accuracy version, with each being between 1.5-3 times more accurate. However, the Midpoint Rule, generally considered the least accurate, was more accurate than any other rule. It was the only rule able to return less than 1% unadjusted error, while others returned errors as high as 30%. It is true that the highest order rules[8] were able to return 0% *adjusted* errors, as opposed to the Midpoint Rule's 0.6%, but the Midpoint is seen to be most consistent, as it does not produce outliers of high error. From this, we can observe that the accuracy of the result is not necessarily correlated with the complexity of the rule, and that often the simpler rules are far more precise.

More significant than the difference in error between rules of different order was the difference in complexity between differing numbers of subdivisions. Using 10^2 subdivisions yielded incredibly high unadjusted errors, as much as 220,000% for Simpson's Rule[9]. Much of

[8] Appendix: Tables 6 and 8
[9] Appendix: Table 7

this resulted from the function x^{1000}, whose integral was incredibly close to 0, and which was badly estimated with the small number of subdivisions. Using 10^5 subdivisions yielded much lower unadjusted errors, with 29.11% being the highest[10], and very low adjusted errors, only as high as 0.688%[11].

In conclusion, while times for all numerical integrations were markedly short, accuracy varied greatly, both between rules and between numbers of subdivisions. While rules of a much larger order are prone to increased error[12], the use of more subdivisions is always possible given a sufficiently powerful computer. Thus, given the current trend towards greater processing power, the use of lower order Newton-Cotes Rules and increased subintervals is remarkably practical.

Gauss-Legendre and Gauss-Kronrod Quadrature

In scientific and mathematical programming libraries, low level algorithms offered can include *Gauss-Legendre Quadrature* and *Gauss-Kronrod Quadrature*. Instead of equally dividing functions into subdivisions, these algorithms evaluate at unequally spaced intervals to reduce the error associated with the higher order Newton-Cotes Rules. In addition, the Gauss-Kronrod method offers the additional benefit of allowing desired error bounds to be directly input, to minimize processing for a desired accuracy, offering a distinct advantage over the previous algorithms which forced the user to guess the inputs needed for an accurate result.

For evaluation, 3-point and a 6-point Gauss-Legendre Rules were used, and error bounds

[10] Appendix: Table 4
[11] Appendix: Table 10

[12] Epperson, James F. "On the Runge Example." The American Mathematical Monthly 94.4 (1987): 329. Web.

of $\pm 10^{-3}$ and $\pm 10^{-6}$ for the Gauss-Kronrod Rule calculations. Neither algorithm was as accurate as the Midpoint Rule performed over 10^5 subdivisions. Rather, the unadjusted percent error was 12.32% and 6.63% for the Gauss-Legendre Rule[13], and 5.05% for both calculations of the Gauss-Kronrod Rule[14]. While the Gauss-Kronrod Rule's adjusted errors or 0.30% were improved, the error from the outlier function was far outside the specified higher accuracy error bounds. The rule also did not improve given stricter error bounds, unlike the Midpoint Rule. The library offered algorithms did have the benefit of being faster with times on the order of 10^{-5} or 10^{-6} seconds for both calculations, but given the scalability and accuracy of the Midpoint Rule, the time difference does not seem enough to make them as practical.

Errors and Functions with Singularities

The issue of the adjusted and unadjusted errors must be considered. Along with the function x^{1000}, the function $1/\sqrt{|x|}$, with an infinite value at x=0, also resulted in high errors for several of the rules. The library algorithms also returned infinite values on up to 2 of the functions with singularities (infinite point values), while the non-library functions did so for none. It is important to note that none of the rules belonging to the "brute force" class are designed to handle functions with such singularities. Rather, they are optimal for functions that can be easily approximated by polynomials, and thus may not be as general purpose as other algorithms. However, the simplicity of the non-library functions allowed them to be adapted to ignore the infinite points and use nearby values to approximate, providing versatility. However, although the returning of a clearly non-real number is problematic, is is less of an issue than large errors, because it merely prompts an examination of the integrated function, rather than

[13] Appendix: Tables 42 and 43
[14] Appendix: Tables 44 and 45

false calculations. In this way, although the Newton-Cotes Rules were able to evaluate a greater number of integrals and the Midpoint Rule accurately calculated the outlier function, the Gauss-Kronrod Rule may offer an advantage here, as it stated that its error for $1/\sqrt{|x|}$ could reach 53.7%. However, this error estimate may be less useful due to its tendency to overestimate the possible error, which in this case was only 1.67%, causing the user to perform unnecessary recalculation.

In conclusion, while brute force algorithms in scientific libraries do have the advantage of faster processing time and, for some, error estimation, they do not necessarily involve a higher degree of accuracy, and may be less effective at calculating integrals of functions with singularities. In terms of error, it was the simplest algorithm that did the best --- the Midpoint Rule also scaled the most in accuracy with increased evaluations, leading it to perhaps be the most effective algorithm of this class due to its performance and extensibility.

Adaptive Algorithms

While many of the algorithms in the first class offered no direct way to input an error goal, calculations can be added to allow for error estimates, and thus error bounds. Such techniques are known as adaptive algorithms, and they involve the recursive selection of which sections of the function to subdivide and more closely evaluate based on previous error estimates [15][16], leading to a greater number of evaluations where the function is rapidly changing, or at singularities. Although simple enough in theory, precise estimates for numerical integration

[15] Gonnet, Pedro. "A Review of Error Estimation in Adaptive Quadrature." CSUR ACM Comput. Surv. ACM Computing Surveys 44.4 (2012): 1-36. Web.
[16] Appendix: adaptiveNonParallel() in MidpointRule.cpp, TrapezoidRule.cpp, SimpsonRule.cpp, Simpson38Rule.cpp, and BoolesRule.cpp

errors often depend on the derivative of the function[17], which is cannot be known if an algorithm is to meet the goal of being general purpose, without regard to the knowledge of the function. However, by comparing less and more accurate intervals on a subinterval, the error can usually be estimated for Newton-Cotes Rules though a simple factor multiplied by the difference between estimates[18]. Other adaptive algorithms available in scientific libraries generally employ similar error estimates. It is important to note, however, that such error estimates are not exact, and depend on estimating how the local error on the subinterval will contribute to the global error.

Adaptive Midpoint and Newton-Cotes Rules

These rules adapt their brute-force counterparts to adapt recursively by dividing subsections in half. $\pm 10^{-3}$ and $\pm 10^{-6}$ were taken as the less and more accurate error bounds, with limits of 10^5 subdivisions on both and of 1 second and 5 seconds of processing time respectively. Generally, the adaptive algorithms proved to be much more accurate than their brute force counterparts. All of the more accurate calculations had unadjusted errors of <6%, with the 2nd and 4th order rules having errors of 0%, with adjusted errors of ≤0.5% for all. The extreme variances between adjusted and unadjusted errors continued for the less accurate calculations, with some unadjusted errors as high as 2700%, and most adjusted errors <1%[19]. This was expected, as the absolute error for the problematic function was inside the given bounds for the less accurate versions. In fact, the majority of the integrations did meet the requested error bounds (more so with the lower order rules), some even exactly, which indicates the accuracy of

[17] Atkinson, Kendall E. "Numerical Integration." An Introduction to Numerical Analysis. New York: Wiley, 1978. 215-28. Print.
[18] Lyness, J. N. "Notes on the Adaptive Simpson Quadrature Routine." Journal of the ACM JACM J. ACM 16.3 (1969): 483-95. Web.
[19] Appendix: Tables 12-21

the simplistic error estimation, and the lack of need for a rigorous, processor heavy mathematical error formula. In the cases where the error estimates did fail, it was often a result of discontinuities that did not immediately appear with 2 subdivisions, leading to errors of 10-20% or higher. The adaptive algorithm in this way was more efficient, but more prone to mistakes than the brute force ones, which consistently evaluated a large number of points. The adaptive algorithm could perhaps be adapted to start their calculations with a larger number of subdivisions (about 10-100), in order to ensure some minimum of accuracy without sacrificing efficiency. Finally, it should be noted that despite the decreased error from the brute force algorithms, there was an increase in functions that could not be evaluated (returning infinity or not a number). Generally it was about 2-4 functions, although some of the less accurate attempts succeeded for all [20].

The average time taken was on the order of 10^{-2} to 1 seconds. Largely, this was the result of several functions taking the maximum time as they attempted to reach the required error goal. The actual time taken for the less accurate calculations usually was either about 1e-6 to 1e-7 seconds or close to 0.5 to 1 seconds, while for the more accurate calculations, it was either about 1e-4 to 1e-5 seconds or close to 5 seconds. In each case, the short times were faster, often much more so, than the equivalent brute force algorithms. While speed may seem to be a disadvantage of the adaptive algorithms, it may in fact be a benefit -- they increase or keep the speed for functions they can accurately integrate, but they also provide information in the form of a time limit or subdivision limit being reached, that can indicate to the user the need for a more accurate calculation by granting more time, or using a different algorithm. This contrasts sharply with the

[20] Appendix: Tables 19 and 21

issue of the brute force algorithms providing answers with completely unknown error values.

Adaptive Gauss-Kronrod Algorithms

Scientific libraries generally provide adaptive algorithms that require varying levels of knowledge of the function to be integrated, from none, to the need for pre-computed approximations of the function. The scope of this essay deals only with algorithms requiring limited to no knowledge in searching for a general purpose algorithm. Three algorithms met this criteria: *Adaptive Gauss-Kronrod Quadrature* and its variations for functions with singularities, either specifically known or not. The most basic of the three is simply an adaptive form of its brute force counterpart, based on comparisons between 15 and 61 point Gauss-Kronrod estimations. The second aims to be more accurate for functions that are known in advance to have singularities, while the third optimizes the second by minimizing run time in exchange for prior knowledge of specific points on the function that are singular.

The rules were evaluated with error bounds of $\pm 10^{-3}$ and $\pm 10^{-6}$ for the less and more accurate computations. In calculating each function except one, the more accurate versions of all three algorithms has percent errors of 0.0%. Even the less accurate versions only had errors of about 0.01% to 0.02% for 1 to 3 functions[21]. The function x^{1000} that many of the algorithms had been unable to compute was estimated to be 0 by many of the algorithms, even when its value (10^{-4}) was within the error bounds. The general Adaptive Gauss-Kronrod rule calculated it without error, and essentially met the goal for accuracy set forth by the research question: it is a general purpose rules that can calculate integrals with little for a large range of functions.

[21] Appendix: Tables 46-51

Because of this high accuracy, it is important to consider speed. Calculation times for these algorithms were on the order of 10^{-5} and 10^{-4} seconds for the less and more accurate computations respectively. Although the algorithm for functions with known singular points was slightly faster than that with unknown points, it was only by a factor of about 1.1 to 1.2, well within the variances in times. This indicates that for most functions, it is unnecessary to have prior knowledge of a function in order to quickly and accurately numerically integrate it.

Adaptive numerical integration algorithms essentially offer a path for computing to improve upon math. The incredibly high accuracy with which the library algorithms consistently and accurately integrated nearly every function in such a short amount of time that 10^5 different functions could be integrated in one second indicates the viability of using a single, general purpose algorithm. The simpler adaptive rules are not without their value however, including being 10 to 100 times faster on certain simpler functions[22], which may be useful when it is generally known that those types of functions are being calculated, and there is not a need for absolute precision. Realistically though, the consistency of the adaptive library algorithms across a range of unknown functions, along with the ease of their implementation makes them more useful than any other type of algorithm.

Parallel Algorithms

While the accurate and fast performance of adaptive library functions may indicate the unnecessary nature of further refinement, the continued progress in parallel computing

[22] Appendix: Tables 12-21

necessitates a close look. One of the great difficulties with parallelizing code is knowing which problems are suited for it and which are not, generally something that is best known by those in the field the problems arise from[23]. Yet on the surface, numerical integration is a problem perfectly suited for parallelization. It involves the repeated and independent calculation of a large number of values, with the goal of improved speed. However, certain classes of numerical integration algorithms may be more suitable for parallelization.

Parallelizing Serial Algorithms

The most important part of parallelizing an algorithm is ensuring that each thread is suitably equal in task load and independent. In other words, it must be ensured that each thread has a *balanced load*[24] so that none spend more time running than any other, while also limiting the *communications overhead* that slows progress while balancing threads. The different classes of algorithms discussed above each have different strengths and weaknesses in the two areas. The non adaptive brute force algorithms are perfectly suited for low communications overhead -- they can utilize *static scheduling* by dividing the region to be integrated into equal parts and having each thread calculate one piece, to be added together at the end. While this essentially ensures zero communications overhead, it has the potential to create an unbalanced load. With the Midpoint and Newton-Cotes rules, the subdivisions are equally spaced, so thread can alternate sections, which balances the load as sections near singularities will take similar amounts of time. The opposite problem is present in parallelizing adaptive code. The adaptive Midpoint and Newton-Cotes rules can be implemented with a queue of subdivisions to be

[23] Wilkes, Maurice V. "The Lure of Parallelism and Its Problems." Computing Perspectives (1995): 75-84. Web.
[24] Matloff, Norman S. "Chapter 2: "Why Is My Program So Slow?": Obstacles to Speed." Parallel Computing for Data Science: With Examples in R, C and CUDA. N.p.: n.p., n.d. 17-34. Print.

integrated, which are then subdivided and returned to the queue if the error is too high. By using *dynamic scheduling* to have all the threads pull from the same queue, load balance should occur. However, this now has the potential for a high amount of communications overhead, as memory protection necessitates none of the threads may add or remove from the queue at the same moment, which can drastically slow down the threads as they get data before each calculation. The library functions are even more difficult to effectively parallelize, because of the inability to implement communication overhead, and because the functions use unequally spaced points, necessitating blocked integration sections, instead of the alternating style of the Newton-Cotes rules. Based on theory, the non library algorithms should be more improved by parallelization, but the extent is difficult to predict.

Effectiveness of Parallelization

Effective parallelization depends on the number of cores of the computer, in this case, 4, allowing for the same number of threads. Despite the theoretical practicalness of parallelizing numerical integration, the reality proved to be quite different. None of the algorithms had consistently reduced time for calculation, and in fact often the time for the parallel code was greater than the time for serial code, likely due to the time for overhead and thread assignments. However, on two or three of the functions, the parallel code ran anywhere from 3-12 times faster than the serial code[25]. This contradicts the theoretical model of 4 times faster execution[26], and may be a result of the large variances in time. Those that ran faster were generally those that approached or reached the maximum number of subdivisions for the adaptive algorithms, or those that contained more complex calculations for each function evaluation for the brute force

[25] Appendix: Tables 22-41 and 52-59
[26] Wilkes, Maurice V. "The Lure of Parallelism and Its Problems." Computing Perspectives (1995): 75-84. Web.

algorithms. These functions also received a boost in speed when run with the library functions, whose loads were not perfectly balanced. While the inconsistency in performance may seem to suggest that parallelization is useless for numerical integration, it is significant that the number of threads was relatively small, where the best that can be hoped for is an integer factor increase in speed[27]. Investigation is warranted into the usefulness of parallelization when utilized by a computer with many more cores.

While many currently working in computer science hope that parallelization will drastically increase processing speed, it is difficult to say whether this is likely. Even on problems like numerical integration that are well suited for parallel algorithms, the advantage is not obvious. Thus, parallelization is not currently significant enough to influence the usefulness of any individual integration algorithm.

Conclusion

While the suitability of different algorithms for numerical integration may initially seem disconnected from the computer being used, and its speed and parallelization ability, questions of such a nature are integral to understanding the previous and future development of such techniques. Some time ago, the difference between simple and complex algorithms, or $\pm 10^{-3}$ and $\pm 10^{-6}$ error bounds may have hinged on one being 10 to 100 times faster than the other, because it could have meant the difference between 1 second and nearly 2 minutes for a calculation. When all calculations are orders of magnitude faster than a second, the difference in accuracy becomes less pronounced. The possibilities of parallelization providing order of magnitude

[27] Wilkes, Maurice V. "The Lure of Parallelism and Its Problems." Computing Perspectives (1995): 75-84. Web.

increases in speed in the future, as opposed to the occasional integer increases of today makes the need for simpler, less accurate, algorithms less and less significant. Algorithms such as Adaptive Gauss-Kronrod Quadrature can now have a high degree of accuracy asked of them, even when not needed, eliminating even the smallest errors. In effect, while the relative accuracy of algorithms might be immutable, their effectiveness and practicality is constantly improving, thus allowing the field of numerical integration itself to improve.

Bibliography

1. Solomon, Justin. "Chapter 14: Integration and Differentiation." Numerical Algorithms: Methods for Computer Vision, Machine Learning, and Graphics. N.p.: n.p., n.d. 277-302. Print.
2. Piessens, R. Quadpack: A Subroutine Package for Automatic Integration. Berlin: Springer-Verlag, 1983. Print.
3. “GNU Scientific Library -- Reference." GNU Scientific Library. Free Software Foundation, Inc., 31 Aug. 2016. Web. 16 Sept. 2016. <https://www.gnu.org/software/gsl/manual/>.
4. Atkinson, Kendall E. "Numerical Integration." An Introduction to Numerical Analysis. New York: Wiley, 1978. 215-28. Print.
5. Epperson, James F. "On the Runge Example." The American Mathematical Monthly 94.4 (1987): 329. Web.
6. Gonnet, Pedro. "A Review of Error Estimation in Adaptive Quadrature." CSUR ACM Comput. Surv. ACM Computing Surveys 44.4 (2012): 1-36. Web.
7. Lyness, J. N. "Notes on the Adaptive Simpson Quadrature Routine." Journal of the ACM JACM J. ACM 16.3 (1969): 483-95. Web.
8. Wilkes, Maurice V. "The Lure of Parallelism and Its Problems." Computing Perspectives (1995): 75-84. Web.
9. Matloff, Norman S. "Chapter 2: "Why Is My Program So Slow?": Obstacles to Speed." Parallel Computing for Data Science: With Examples in R, C and CUDA. N.p.: n.p., n.d. 17-34. Print.

Appendix

Figure 1 - Tested Functions

Simple Functions:	*Discontinuous Functions:*	*Badly Behaved Functions:*
1. $\int_0^1 sin(x)dx$	9. $\int_0^1 \frac{e^x-1}{x}dx$	17. $\int_0^{\pi} e^x ln(sin(x))dx$
2. $\int_0^1 x^2 dx$	10. $\int_{-9}^1 \frac{dx}{\sqrt{\lvert x\rvert}}$	18. $\int_0^1 ln(x)dx$
3. $\int_0^1 x^3 dx$	11. $\int_0^1 sin(x)(\sqrt{1-x^2})dx$	19. $\int_8^9 ln(x^2)dx$
4. $\int_0^1 x^{10000} dx$		20. $\int_0^1 ln(\frac{1}{x})dx$
5. $\int_{-20}^0 (3x^4 + 4x^3 + 76x^2 + 58x + 4)dx$	12. $\int_0^3 \frac{sin(x)}{x}dx$	21. $\int_0^{\frac{2}{\pi}} x^2 sin(\frac{1}{x})dx$
6. $\int_1^2 \frac{dx}{x}$	13. $\int_0^1 \sqrt{\lvert x-0.7\rvert}dx$	22. $\int_0^1 \frac{\sqrt{1-x^4}}{x^{1-1/\pi}}dx$
7. $\int_0^1 e^x dx$	14. $\int_1^3 \frac{(x+2)(x-2)}{(x-2)}dx$	23. $\int_0^1 \frac{e^x}{x^{1/\pi}}dx$
8. $\int_0^1 \sqrt{x}dx$	15. $\int_0^2 [x]\,dx$	
	16. $\int_0^2 f(x)dx$	
	$f(x) = \begin{cases} \frac{x}{2} & x < 1 \\ \frac{3x}{2} & x > 1 \end{cases}$	