Technical Report

TR-2008-012

Iterative Adaptive Simpson and Lobatto Quadrature in Matlab

by

Matthias Conrad, Nils Papenberg

Mathematics and Computer Science

EMORY UNIVERSITY

Iterative Adaptive Simpson and Lobatto Quadrature in MATLAB

Matthias Conrad* and Nils Papenberg

July 1, 2008

Abstract

Numerical integration methods are discussed in almost every standard book on numerics. In these books quadrature algorithms like the adaptive Simpson or the adaptive Lobatto algorithm are presented in a recursive way. The benefit of the recursive programming is the compact and clear representation. However, recursive quadrature algorithms might be transformed into iterative quadrature algorithms without major modifications in the structure of the algorithm. We present a general iterative adaptive quadrature algorithm, which preserves the compactness and the clarity of recursive algorithms. Our iterative algorithm provides a parallel calculation of the integration function. We introduce a Matlab implementation of the adaptive Simpson and the adaptive Lobatto of our algorithm and show results, which indicates a tremendous gain in run-time. Our results suggest a general iterative and not a recursive implementation of adaptive quadrature formulas, once the programming language permits parallel access to the integration function.

1 Introduction

Let the functions $f^{(k)} : [a, b] \to \mathbb{R}$, $k = 1, \ldots, n$ are given on a bounded real interval $[a, b]$. In the following we are interested to compute numerically the

*corresponding author: Matthias Conrad, Department of Mathematics and Computer Science, Emory University, 400 Dowman Drive E429, 30322 Atlanta, GA, USA, e-mail: conrad@mathcs.emory.edu

integrals

$$I(k) = \int_a^b f^{(k)}(t)\mathrm{d}t \qquad k = 1, \ldots, n$$

simultaneously. Therefore we will present two quadrature formulas in this work: on the one hand the often used adaptive Simpson and on the other hand the also well known adaptive Lobatto procedure (for detailed informations and advanced readings see [DR84; GG00; KU94]). Our implementations are based upon the functions `quad` and `quadl` for $n = 1$ integrated in MATLAB. These implementations again depend on algorithms of Gander and Gautschi published in 2000 [GG00]. The goal of our algorithmic implementation was to obtain a run-time improvement (special for $n \geq 1$) in relation to the function `quadv` – a vectorial version of the function `quad`. The authors are aware of the fact that the adaptive Simpson and the adaptive Lobatto procedure can not compete with sophisticated state of the art quadrature algorithms. However, based on the same algorithm our implementations `adaptiveSimpson` and `adaptiveLobatto` show better performances than the standard MATLAB functions `quad`, `quadl`, and `quadv`, in general (see section 3).
Note, it is impossible to construct a numerical integration algorithm which is foolproof. For each numerical integration algorithm it is easy to construct a function f for which this algorithm will not integrate correctly [Kah80; dB71].

2 Methods

Commonly adaptive quadrature procedures are implemented as a recursive process (e.g. see [DR84]). Our new algorithmic implementation transfers this recursive process of the adaptive quadrature into an *iterative* method. The advantage of the iterative method is deeply rooted in the parallelization of refinement steps and the efficiently utilization of MATLABs vectorization. The number of function accesses of f is reduced, in general (see Figure 3). Based on this iterative structure the functional calls can be parallel accomplished in the algorithmic implementation; MATLAB in particular can use this parallelization efficiently, while a recursive implementation requires however sequentially more frequent function calls (see Figure 3).
As a further small change compared to the algorithms `quad`, `quadl`, and `quadv` integrated in MATLAB, we implemented a pole detection, as well as an integration over an infinite intervals, speak $a \in \mathbb{R} \cup \infty$ and $b \in \mathbb{R} \cup \infty$.

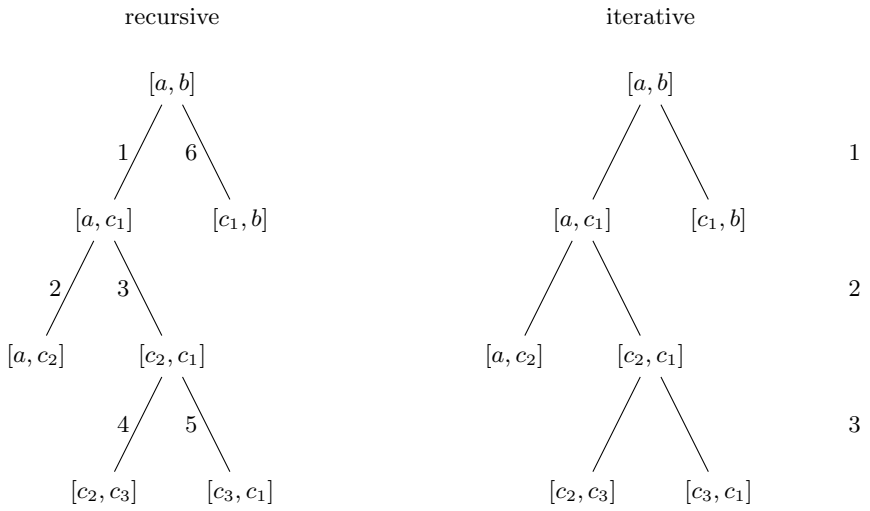

Figure 1: While in this example the recursive algorithm requires six functional calls for an adaptive refinement of the interval $[a, b]$, the iterative algorithm needs only three calls of the function. Note, that the number of function evaluation stays the same.

The integration on infinite intervals is carried out by a simple substitution

$$\int_a^b f(t)\mathrm{d}t = \begin{cases} \int_0^1 \frac{f(\log t)}{t}\mathrm{d}t & \text{for } a = -\infty, b = 0, \\ \int_0^1 \frac{f(-\log t)}{t}\mathrm{d}t & \text{for } a = 0, b = \infty, \\ \int_{-\pi/2}^{\pi/2} \frac{f(\tan t)}{\cos^2 t}\mathrm{d}t & \text{for } a = -\infty, b = \infty. \end{cases}$$

This substitution allows to calculate simple numeric integrations of frequently used distribution functions as Gauss or gamma distribution (see for example test-functions 27). It should be mentioned that such a simple substitution has its limitation and the user should be aware of this cutbacks. See [DR84, Chapter 3] for further readings how to handle infinite intervals and singularities. We examined our new algorithms `adaptiveSimpson` and `adaptiveLobatto` (for the implementations see appendix B) with various test-functions (see section A). The test-functions 1 to 21 are classical test-functions and are published by Kahaner in 1971 [Kah71]. Moreover, we added several test-function for further investigations. The test-functions can be roughly divided in six classes (see for example [Esp07]):

1. continuous functions (c),

2. discontinuous functions (d),

3. oscillating functions (o),

4. functions with singularities (s),

5. functions on infinite intervals (i),

6. functions with peaks (p).

Depending upon the class of function this may lead to significant run-time improvements (see Figure 4).

2.1 Algorithm

In order to avoid indices, we present the algorithms only for $n = 1$ ($f = f^{(1)}$). A vectorial representation can be accomplished and tested similarly.

Algorithm 1 Iterative adaptive quadrature algorithm

Input: function $f : [a, b] \to \mathbb{R}$

Output: approximation integral $Q \approx \int_a^b f(t)\ \mathrm{d}t$

1: $Q \leftarrow 0$
2: $J \leftarrow \{[a, b]\}$

3: **repeat**
4: calculate Q_I^1, Q_I^2 with quadrature formulas for all $I \in J$
5: $\bar{J} \leftarrow \{I \in J\ :\ |Q_I^1 - Q_I^2| < \delta\}$

6: calculate $Q_{\bar{I}}$ with quadrature formula for all $\bar{I} \in \bar{J}$
7: $Q \leftarrow Q + \sum_{\bar{I} \in \bar{J}} Q_{\bar{I}}$

8: $J \leftarrow J \setminus \bar{J}$
9: for all $I \in J$ remove I out of J and replace I by a partition of I
10: **until** $J = \emptyset$

11: **return** Q

The pseudo-code of the iterative adaptive quadrature algorithm is shown in algorithm 1. For an easier understanding we give some detailed remarks on the algorithm:

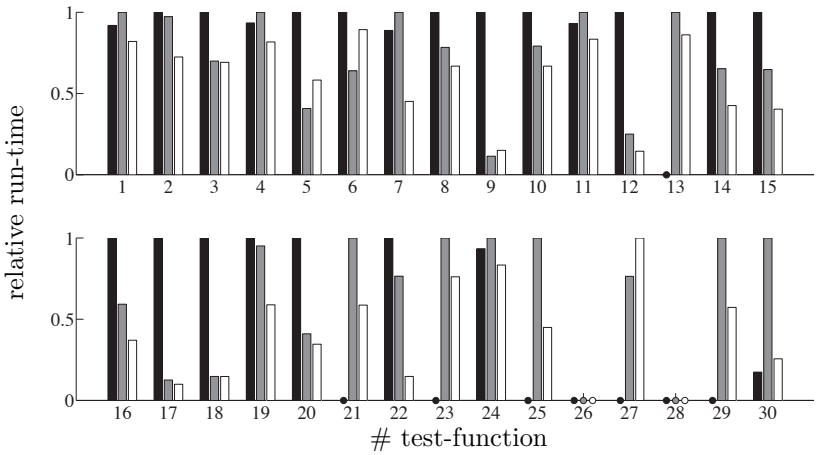

Figure 2: The timings of our new algorithms `adaptiveSimpson` (gray) and `adaptiveLobatto` (white) compared with the MATLAB function `quadv` (black). Tested on the test-functions 1 – 30 of appendix A with a tolerance (`tol`) $\delta = 1 \cdot 10^{-6}$ and `parts` $= 10$ for `adaptiveSimpson` and `parts` $= 5$ for `adaptiveLobatto`. The timing is given relative to the slowest algorithm.

1. The set J contains subsets of the interval $[a, b]$, for which the numerical integrations do not fulfill a given tolerance. In the initial state J contains the complete interval $[a, b]$.

2. The values of Q_I^1 and Q_I^2 in line 4 are obtained by quadrature formulas of different order for each subinterval I out of J. Intervals, where the difference $|Q_I^1 - Q_I^2|$ is smaller than a given tolerance δ, are collected in the set $\bar{J}$ (line 5). For all of these subintervals $\bar{I} \in \bar{J}$ the numerical integration values $Q_{\bar{I}}$ are calculated (usually out of $Q_{\bar{I}}^1$ and $Q_{\bar{I}}^2$, see [GG00]), which are added on the total quadrature value Q (lines 6f.). The subintervals $\bar{I}$ are removed from J (line 8).

3. The subintervals remaining in the set J are replaced by partitions (line 9), (example: the interval $[a, x_0]$ replaced by $[a, \frac{a+x_0}{2}]$ and $[\frac{a+x_0}{2}, x_0]$).

4. The function f is called only once in line 4, while the number function evaluations depend on the quadrature formula and the size of I.

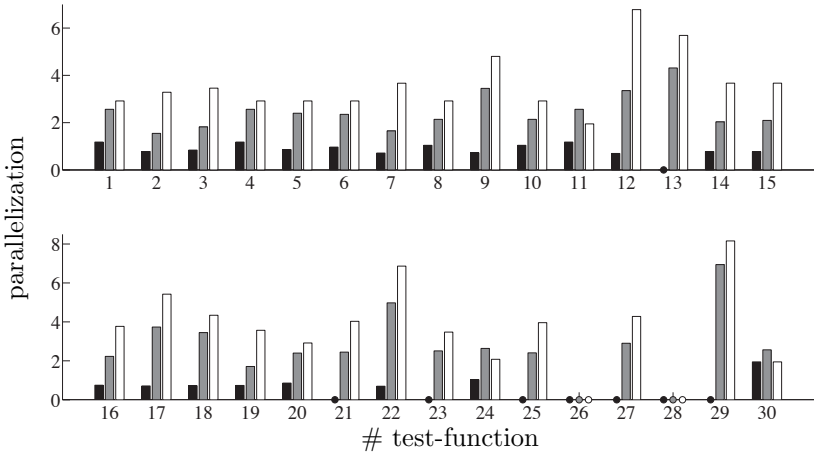

Figure 3: The degree of parallelization of the algorithms on the test-functions. For the ad-measurement of the parallization of the algorithms we choose the logarithm of the number of function evaluations divided by the number of function calls.

3 Results & Disscussion

First it should be mentioned that the presented iterative adaptive Simpson quadrature `adaptiveSimpson` does not differ importantly from the MATLAB function `quadv`. However, our new algorithm performs substantially better in run-times. The results of our calculations on the test-function of appendix A can be seen in Table 3 and are illustrated in the Figures 2, 3, and 4. The largest differences in run-time can be seen in oscillating functions like the test-function # 9, 17, and 18. Since the curvature of these functions varies often, the numerical integration (e.g. Simpson formula) often performs an inadequate approximation and the interval must be refined in many cases. This leads to many sequential function calls in the recursive algorithm, while the iterative algorithm benefits by parallel refinements. Even despite a higher number of function evaluation, the iterative algorithm may reduce the run-time and the function calls in relation to the recursive algorithms (see table 3, e.g. test-function # 17). The user of our algorithms `adaptiveSimpson` and `adaptiveLobatto` may choose the degree of initial partitioning of the interval $[a, b]$ by changing the parameter `parts` (default is `parts` $= 2$, which corresponds to the whole interval $[a, b]$). Therefore, the user may reduce the iteration depth by choosing a finer partition and this may lead to shorter run-times (see Figure 4). Note, an improvement of the accuracy of the numerical quadrature comes along generally with an increase

of the initial partition. All calculations are accomplished on a MacBook with 2.16 GHz Intel Core 2 Duo processor with 2 GB RAM.

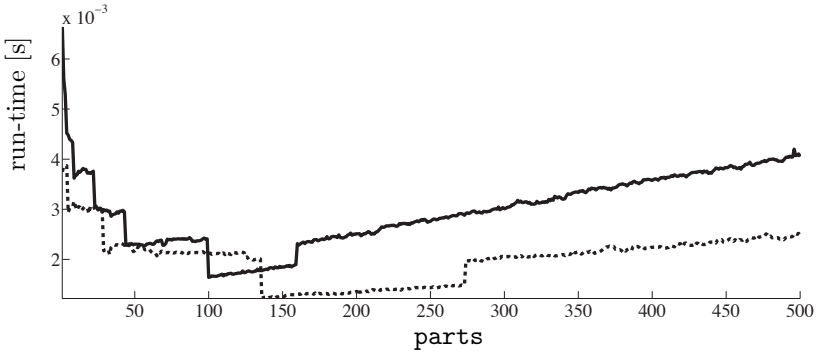

Figure 4: The solid curve corresponds to the adaptive Simpson and the dashed curve to the adaptive Lobatto algorithm. Here we use the test-function # 17. The calculation for each value of `parts` is repeated 2000 times and the mean value of the run-times are show.

For a comparison of the number of function evaluation in relation to the obtained accuracy with other quadrature packages can be found in [GG00]. Our iterative algorithms `adaptiveSimpson` and `adaptiveLobatto` are superior to the recursive algorithms `quad`, `quadl`, and `quadv` in many test cases. With these new algorithms we are able to compute numerically the integral of various classes of functions fast and efficiently. In addition these algorithms can be adapted to special requirements quite easily. One goal could be to calculate functions with poles by splitting the intervals at poles and calculate these intervals separately (see [DR84, Chapter 2]). In our algorithms we only apply a rudimentary substitution of function on unrestricted intervals, here more sophisticated methods can be integrated [DR84, Chapter 3]. A very promising extension of our algorithm would be the paralleled computation of functions $f^{(j)}$, $j = 1, \ldots, n$ on different intervals $[a^{(j)}, b^{(j)}]$, $j = 1, \ldots, n$.

Acknowledgment: The authors like to thank Bernd Fischer and Jan Modersitzki as well as the entire SAFIR group of the Institute of Mathematics at the University of Lübeck for their constructive feedback.

#		run-time [ms]	fcn evals	fcn calls	error	#		run-time [ms]	fcn evals	fcn calls	error
	m	1.30	26	11	3.743e-09		m	9.94	186	91	3.983e-07
1	s	1.42	242	1	3.997e-15	16	s	5.89	346	7	3.022e-07
	l	1.16	70	1	$\star$		l	3.68	430	4	9.377e-08
	m	6.53	122	59	1.229e-05		m	32.12	586	291	3.052e-05
2	s	6.35	306	9	8.251e-06	17	s	4.04	674	4	2.017e-06
	l	4.73	370	6	8.301e-08		l	3.21	1750	3	2.938e-08
	m	3.70	74	35	7.097e-06		m	20.74	378	187	1.490e-08
3	s	2.59	258	3	5.047e-06	18	s	3.07	346	3	4.592e-07
	l	2.56	190	3	1.036e-07		l	3.07	550	3	4.972e-08
	m	1.28	26	11	5.549e-07		m	10.32	188	92	6.870e-06
4	s	1.37	242	1	5.559e-13	19	s	9.82	348	13	1.062e-05
	l	1.12	70	1	3.331e-16		l	6.08	552	7	6.520e-10
	m	3.32	66	31	8.213e-08		m	3.59	66	31	1.519e-08
5	s	1.35	242	1	2.252e-11	20	s	1.48	242	1	4.289e-12
	l	1.93	190	2	1.626e-07		l	1.24	70	1	5.401e-12
	m	2.11	42	19	2.068e-06		m	9.01	162	79	1.065e-03 ‡
6	s	1.35	242	1	3.773e-08	21	s	5.47	354	6	2.193e-06
	l	1.88	130	2	4.464e-11		l	3.21	670	3	5.106e-08
	m	20.01	396	196	1.061e-05		m	206.24	3706	1851	†
7	s	22.52	564	35	8.318e-06	22	s	157.79	40426	125	†
	l	10.17	1032	13	1.401e-07		l	30.50	45970	19	†
	m	1.73	34	15	9.835e-08		m	4.06	74	35	†
8	s	1.36	242	1	1.299e-14	23	s	3.93	338	4	†
	l	1.16	70	1	2.220e-14		l	2.99	490	3	†
	m	19.61	378	187	5.027e-07		m	1.48	28	12	2.906e-10
9	s	2.24	282	2	3.165e-07	24	s	1.59	244	1	8.882e-16
	l	2.95	970	3	9.215e-08		l	1.33	72	1	8.882e-16
	m	1.78	34	15	1.930e-08		m	27.15	490	243	5.503e-04 ‡
10	s	1.41	242	1	2.626e-13	25	s	27.80	898	36	1.709e-05
	l	1.19	70	1	2.776e-15		l	12.49	2050	14	2.685e-07
	m	1.35	26	11	8.558e-10		m	3.20	58	27	‡
11	s	1.45	242	1	6.106e-16	26	s	3.07	274	3	‡
	l	1.21	70	1	$\star$		l	2.16	250	2	‡
	m	189.86	3524	1760	†		m	1.38	26	11	‡
12	s	47.53	3908	58	†	27	s	3.04	258	2	1.980e-06
	l	27.52	39312	21	†		l	3.98	490	3	1.012e-07
	m	64.94	1194	595	5.393e-02 ‡		m	1.42	26	11	‡
13	s	5.05	1354	5	1.205e-07	28	s	58.99	1156	63	‡
	l	4.35	3370	4	1.214e-08		l	364.03	72494	304	‡
	m	6.63	122	59	1.772e-08		m	1.57	30	13	†
14	s	4.33	306	5	1.788e-06	29	s	55.42	49554	19	†
	l	2.83	250	3	9.245e-13		l	31.74	44050	8	†
	m	6.99	130	63	4.941e-07		m	0.92	14	5	$\star$
15	s	4.53	338	5	1.200e-07	30	s	5.31	242	1	$\star$
	l	2.83	250	3	9.355e-09		l	1.36	70	1	$\star$

Table 1: This table corresponds to the results of the 30 test-function (with id number #) of section A. We compare the results of the algorithms `quadv` (m) and our new algorithms `adaptiveSimpson` (s) and `adaptiveLobatto` (l). The calculation for each test-function is repeated 2000 times. The *run-times* are shown in milli-seconds. The number of evaluation points is shown in the column *fcn evals*, while the number of function calls are shown in *fcn calls*. Furthermore we show the *error* between the analytical solution of the integral and the calculated quadrature, here ($\star$) indicates that the error of the quadrature formula is below the machine accuracy. The (†) indicates, that the analytic solution of the integral is not finite and (‡) signifies the failure of the precision of the algorithm (plus tolerance $1 \cdot 10^{-5}$ for a given tolerance value $\delta = 1 \cdot 10^{-6}$). Note, the MATLAB function `quadl` fails on test-function # 30.

A Test-Functions

#	$f(t)$	I	$\int_a^b f(t)\mathrm{d}t$	class
1	e^t	$[0,1]$	$\mathrm{e}-1$	c
2	$\begin{cases}1, & \text{for } t \geq 0.3\\ 0, & \text{else}\end{cases}$	$[0,1]$	0.7	d
3	$\sqrt{t}$	$[0,1]$	$2/3$	c
4	$0.92\cosh t - \cos(t)$	$[-1,1]$	$1.84\sinh(1) - 2\sin(1)$	c
5	$\frac{1}{t^4+t^2+0.9}$	$[-1,1]$	$0.158223296372967 \times 10^1$	c
6	$t^{3/2}$	$[0,1]$	0.4	c
7	$\frac{1}{\sqrt{t}}$	$[0,1]$	2	s
8	$\frac{1}{t^4+1}$	$[0,1]$	0.866972987339911	c
9	$\frac{2}{2+\sin(10\pi t)}$	$[0,1]$	$\frac{2}{\sqrt{3}}$	o
10	$\frac{1}{1+t}$	$[0,1]$	$\log(2)$	c
11	$\frac{1}{1+\mathrm{e}^t}$	$[0,1]$	$\frac{\ln(\mathrm{e})}{\mathrm{e}+1} - \ln(0.5)$	c
12	$\frac{1}{\mathrm{e}^t-1}$	$[0,1]$	∞	s
13	$\frac{\sin 100\pi t}{\pi t}$	$[0.1,1]$	$0.909863753916684 \times 10^{-2}$	o
14	$\sqrt{50}\,\mathrm{e}^{-50\pi t^2}$	$[0,10]$	0.5	p
15	$25\,\mathrm{e}^{-25t}$	$[0,10]$	1	p
16	$\frac{50}{\pi(2500t^2+1)}$	$[0,10]$	$\frac{1}{\pi}\arctan(500)$	p
17	$\frac{50(\sin 50\pi t)^2}{(50\pi t)^2}$	$[0.01,1]$	0.112139303741637	o
18	$\cos(\cos t + 3\sin t + 2\cos 2t + 3\sin 2t + 3\cos 3t)$	$[0,4]$	0.966440320387790	o
19	$\log t$	$[0,1]$	-1	s
20	$\frac{1}{t^2+1.005}$	$[0,1]$	$\frac{4}{\sqrt{4.02}}\arctan\left(\frac{2}{\sqrt{4.02}}\right)$	c
21	$\frac{1}{(\cosh 10(t-0.2))^2} + \frac{1}{(\cosh 100(t-0.4))^4} + \frac{1}{(\cosh 1000(t-0.6))^6}$	$[0,1]$	0.210802735500549	p
22	$\frac{1}{\lvert t-0.3\rvert}$	$[0,1]$	$-\infty$	s
23	$\frac{1}{\lvert t-0.5\rvert}$	$[0,1]$	$-\infty$	s
24	$\frac{\sin(t)}{t}$	$[0,1]$	0.946083070367182	s
25	$\frac{1}{\sqrt{\lvert t-0.3\rvert}}$	$[0,1]$	$2(\sqrt{0.3}+\sqrt{0.7})$	s
26	$\frac{1}{\sqrt{\lvert t-0.5\rvert}}$	$[0,1]$	$2\sqrt{0.5}$	s
27	$\frac{1}{\sqrt{2\pi}}\,\mathrm{e}^{-\frac{1}{2}t^2}$	$[-\infty,\infty]$	1	i
28	$\frac{\sin(t)\cdot\cos(0.1t)}{t}$	$[0,\infty]$	$\pi/2$	i
29	t^2	$[-\infty,\infty]$	∞	i
30	1	$[0,0]$	0	s

B Matlab code

B.1 adaptiveSimpson.m

```
function [Q fcnEvals iter] = adaptiveSimpson(fcn, a, b, varargin)
%
% function [Q fcnEvals] = adaptiveSimpson(fcn, a, b, varargin)
%
% (c) Matthias Conrad and Nils Papenberg in August 2007
%     contact: conrad@tiaco.de
%
% Version (changes):
%   1.0 first final version 2007-08-03
%
% Description:
%   adaptive Simpson algorithm programmed in an iterative not recursive
%   manner
%
% Input arguments:
%   fcn             - function to be integrated
%   a               - first point of interval
%   b               - final point of interval
%   #varargin       - further options of algorithm
%     tol           - tolerance accuracy of quadrature
%     parts         - initial number of partitions
%     maxFcnEvals   - maximal number of function evaluations allowed
%     maxPartitions - maximal number of partitions allowed
%
% Output arguments:
%   Q               - numerical integral of function fcn on [a,b]
%   fcnEvals        - number of function evaluations
%   iter            - number of iterations
%
% Details:
%   This function uses elements of Matlab integrated function "quad".
%
% References:
%   [1] Gander, W. & Gautschi, W. Adaptive Quadrature - Revisited
%       Eidgenoessische technische Hochschule Zuerich, 2000.

% check scalar limits of interval
if ~isscalar(a) || ~isscalar(b),
  error('Matlab:adaptiveSimpson:Limits',...
    'The limits of integration must be scalars.');
end;

% default values
tol = 1e-6; parts = 2; maxFcnEvals = 20000; maxParts = 8000;

% rewrite default options if needed
for j = 1 : length(varargin) / 2,
  eval([varargin{2 * j - 1},'=varargin{',int2str(2 * j),'};']);
end;

% initial values, termination constant (incl. quadr factor 15), parts of interval and integral value
tol = 15 * tol; m = parts;  parts = 4 * parts + 1; Q = 0;
```

```
iter = 0; fcnEvals = 0; maxResolution = 0; minH = eps(b - a) / 1024;
poleWarning = 0;

% check if interval has infinite boundaries, in case substitute function
if ~isfinite(a) || ~isfinite(b),
  warning('Matlab:adaptiveSimpson:infiniteInterval',...
    'The integral has an infinite interval; proceed with a substitution of function on finite interval.')
  if ~isfinite(a) && isfinite(b),
    [Q fcnEvals iter] = adaptiveSimpson(fcn, 0, b, varargin);
    fcn = @(t) infiniteLeft(t, fcn);
    a = 0; b = 1;
  elseif isfinite(a) && ~isfinite(b),
    [Q fcnEvals iter] = adaptiveSimpson(fcn, a, 0, varargin);
    fcn = @(t) infiniteRight(t, fcn);
    a = 0; b = 1;
  else,
    fcn = @(t) infiniteBoth(t, fcn);
    a = - pi / 2; b = pi / 2;
  end;
end;

% choice of initial evaluation points
t = [a + (b - a) / m * (kron(ones(1,m), [0, 0.27158, 0.72842]) + kron(0: m - 1, [1 1 1])), b];
H = diff(t);
t = [t(1:end-1); t(1:end-1) + H/4; t(1:end-1) + H/2; t(1:end-1) + 3 * H/4;];
t = [t(:);b]';

% initialize equidistant mesh and dimension of fcn
y = fcn(t); fcnEvals = fcnEvals + length(y); n = size(y,1);

% avoid infinities at start point of interval
if any(~isfinite(y(:,1))),
  y(:,1) = fcn(a + eps(superiorfloat(a,b)) * (b - a));
  fcnEvals = fcnEvals + 1;
  poleWarning = 1;
end;

% avoid infinities at end point of interval
if any(~isfinite(y(:, end))),
  poleWarning = 1;
  y(:, end) = fcn(b - eps(superiorfloat(a,b)) * (b - a));
  fcnEvals = fcnEvals + 1;
  poleWarning = 1;
end;

% poles at initial points
if ~isempty(find(~isfinite(max(abs(y))))), poleWarning = 1; end;

% initialize interval boundaries and values
A = t(1:4:end-1); yA = y(:, 1:4:end-1);
B = t(5:4:end);   yB = y(:, 5:4:end);
C = t(3:4:end-1); yC = y(:, 3:4:end-1);
D = t(2:4:end-1); yD = y(:, 2:4:end-1);
E = t(4:4:end-1); yE = y(:, 4:4:end-1);

% adaptive Simpson iteration
```

```
while 1,

  % number of iteration
  iter = iter + 1;

  % Simpson formulas (on rough and fine grid)
  Q1 = kron(H, ones(n,1)) / 6 .* (yA + 4 * yC + yB);
  Q2 = kron(H, ones(n,1)) / 12 .* (yA + 4 * yD + 2 * yC + 4 * yE + yB);

  % difference of Simpson formulas
  diffQ = Q2 - Q1; diffQ(find(isnan(diffQ))) = 0;

  % intervals which do not fulfill termination criterion
  idx = find(max(abs(diffQ), [], 1) > tol);

  % intervals fulfill termination criterion
  idxQ = setdiff(1:length(A), idx);

  % check stop criterions
  STOP1 = isempty(idx); % check regular termination
  STOP2 = fcnEvals > maxFcnEvals;  % check maximal function evaluations
  STOP3 = 2 * length(idx) > maxParts; % check maximal partition

  % regular termination
  if STOP1,
    Q = Q + sum(Q2 + diffQ / 15, 2);
    break;
  end;

  % check if maximal resolution reached
  idxH = find(abs(H) < minH);
  if ~isempty(idxH),
    Q = Q + sum(Q2(idxH) + diffQ(idxH) / 15, 2);
    idx = setdiff(idx, idxH);
    idxQ = setdiff(idxQ, idxH);
    maxResolution = 1;
    % termination criterion
    if isempty(idx), break, end;
  end;

  % maximal function evaluations reached
  if STOP2,
    warning('Matlab:adaptiveSimpson:MaxEvaluations',...
      'The maximal number of function evaluations reached; singularity likely.')
    Q = Q + sum(Q2 + diffQ / 15, 2);
    break;
  end;

  % maximal partition reached
  if STOP3,
    warning('Matlab:adaptiveSimpson:parts',...
      'The maximal number of parts reached.')
    Q = Q + sum(Q2 + diffQ / 15, 2);
    break;
  end;
```

```
  % update quadrature value
  Q = Q + sum(Q2(:,idxQ) + diffQ(:,idxQ) / 15, 2);

  % update A, B, and C
  t = zeros(1, 2*length(idx));
  t(1:2:end) = A(idx); t(2:2:end) = C(idx); A = t;
  t(1:2:end) = C(idx); t(2:2:end) = B(idx); B = t;
  t(1:2:end) = D(idx); t(2:2:end) = E(idx); C = t;

  % update yA, yB, and yC
  y = zeros(n, 2*length(idx));
  y(:,1:2:end) = yA(:,idx); y(:,2:2:end) = yC(:,idx); yA = y;
  y(:,1:2:end) = yC(:,idx); y(:,2:2:end) = yB(:,idx); yB = y;
  y(:,1:2:end) = yD(:,idx); y(:,2:2:end) = yE(:,idx); yC = y;

  % update interval length
  H = B - A;

  % update D and E by interval bisection
  D = (A + H / 4); E = (A + 3 * H / 4);

  % calculate yD and yE
  y = fcn([D,E]); fcnEvals = fcnEvals + 4 * length(idx);

  % poles at new points
  if ~isempty(find(~isfinite(max(abs(y))))), poleWarning = 1; end

  % assign new values ob yD and yE
  yD = y(:,1:end/2); yE = y(:,end/2+1: end);

end;

% display warnings
if any(~isfinite(Q)),
  warning('Matlab:adaptiveSimpson:Infinite',...
    'The Quadrature of the function reached infinity or is Not-a-Number.')
end;
if maxResolution,
  warning('Matlab:adaptiveSimpson:MaxResolution',...
    'The maximal resolution of partial interval reached; singularity likely.')
end;
if poleWarning,
  warning('Matlab:adaptiveSimpson:PoleDetection',...
        'A detection of a pole; singularity likely.')
end;
return;

% substitute function interval [0, inf] on [0, 1]
function f = infiniteLeft(t, fcn)
f = fcn(log(t)) ;
f = f ./ kron(ones(size(f,1),1), t);
return;

% substitute function interval [-inf, 0] on [0, 1]
function f = infiniteRight(t, fcn)
f = fcn(-log(t));
```

```
f = f ./ kron(ones(size(f,1),1), t);
return;

% substitute function interval [-inf, inf] on [-pi / 2, pi / 2]
function f = infiniteBoth(t, fcn)
f = fcn(tan(t));
f = f ./ kron(ones(size(f,1),1), cos(t).^2);
return;
```

B.2 adaptiveLobatto.m

```
function [Q fcnEvals iter] = adaptiveLobatto(fcn, a, b, varargin)
%
% function [Q fcnEvals] = adaptiveLobatto(fcn, a, b, varargin)
%
% (c) Matthias Conrad and Nils Papenberg in August 2007
%     contact: conrad@tiaco.de
%
% Version (changes):
%   1.0 first final version 2007-08-03
%
% Description:
%   adaptive Lobatto algorithm programmed in an iterative not recursive
%   manner
%
% Input arguments:
%   fcn             - function to be integrated
%   a               - first point of interval
%   b               - final point of interval
%   #varargin       - further options of algorithm
%     tol           - tolerance accuracy of quadrature
%     parts         - initial number of partitions
%     maxFcnEvals   - maximal number of function evaluations allowed
%     maxPartitions - maximal number of partitions allowed
%
% Output arguments:
%   Q               - numerical integral of function fcn on [a,b]
%   fcnEvals        - number of function evaluations
%   iter            - number of iterations
%
% Details:
%   This function uses elements of Matlab integrated function "quadl".
%
% References:
%   [1] Gander, W. & Gautschi, W. Adaptive Quadrature - Revisited
%       Eidgenoessische technische Hochschule Zuerich, 2000.

% check scalar limits of interval
if ~isscalar(a) || ~isscalar(b),
  error('Matlab:adaptiveLobatto:Limits',...
    'The limits of integration must be scalars.');
end;

% default values
tol = 1e-6; parts = 2; maxFcnEvals = 20000; maxParts = 8000;

% rewrite default options if needed
```

```
for j = 1 : length(varargin) / 2,
eval([varargin{2 * j - 1},'=varargin{',int2str(2 * j),'};']);
end;

% initial values, termination constant, parts of interval and integral value
m = parts; parts = 4 * parts + 1; Q = 0;
minH = eps(b - a) / 1024; maxResolutionWarning = false; iter = 0;
poleWarning = 0;

% width constants
alpha = sqrt(2/3); beta = sqrt(1/5);

% check if interval has infinite boundaries, in case substitute function
if ~isfinite(a) || ~isfinite(b),
  warning('Matlab:adaptiveLobatto:infiniteInterval',...
    'The integral has an infinite interval; proceed with a substitution of function on finite interval.')
  if ~isfinite(a) && isfinite(b),
    [Q fcnEvals iter] = adaptiveLobatto(fcn, 0, b, varargin);
    fcn = @(t) infiniteLeft(t, fcn);
    a = 0; b = 1;
  elseif isfinite(a) && ~isfinite(b),
    [Q fcnEvals iter] = adaptiveLobatto(fcn, a, 0, varargin);
    fcn = @(t) infiniteRight(t, fcn);
    a = 0; b = 1;
  else,
    fcn = @(t) infiniteBoth(t, fcn);
    a = - pi / 2; b = pi / 2;
  end;
end;

% initialize grid
t = linspace(a, b, m + 1);
A = t(1:end-1); B = t(2:end);

% widths and midpoints of intervals
H = diff(t)/2; J = (A + B) / 2;

% grid points
F = -alpha * H + J; D = -beta * H + J; C =  J;
E =  beta * H + J; G =  alpha * H + J;
t = [A; F; D; C; E; G; B]; t = t(:);

% function evaluations
y = fcn([A, F, D, C, E, G, B]); fcnEvals = 7 * m;

% avoid infinities at start point of interval
if any(~isfinite(y(:,1))),
y(:,1) = fcn(a + eps(superiorfloat(a,b)) * (b - a));
  fcnEvals = fcnEvals + 1;
end;

% avoid infinities at end point of interval
if any(~isfinite(y(:, end))),
y(:, end) = fcn(b - eps(superiorfloat(a,b)) * (b - a));
fcnEvals = fcnEvals + 1;
end;
```

```
% poles at initial points
if ~isempty(find(~isfinite(max(abs(y))))), poleWarning = 1; end;

% hand over function values
yA = y(:,     1 :   m); yF = y(:,   m+1 : 2*m); yD = y(:, 2*m+1 : 3*m);
yC = y(:, 3*m+1 : 4*m); yE = y(:, 4*m+1 : 5*m); yG = y(:, 5*m+1 : 6*m);
yB = y(:, 6*m+1 : end);

% dimension of parallel integration
n = size(yA,1);

% adaptive Lobatto iteration
while 1,

  % number of iteration
  iter = iter + 1;

  % four point Lobatto formula
  Q1 = kron(H, ones(n,1)) / 6 .* (yA + 5 * (yD + yE) + yB);
  % seven point Kronrod formula
  Q2 = kron(H, ones(n,1)) / 1470 .* (77 * (yA + yB) + 432 * (yF + yG) + 625 * (yD + yE) + 672 * yC);

  % difference of Lobatto formulas
  diffQ = Q2 - Q1; diffQ(find(isnan(diffQ))) = 0;

  % intervals which do not fulfill termination criterion
  idx = find(max(abs(diffQ), [], 1) > tol);

  % intervals fulfill termination criterion
  idxQ = setdiff(1:length(A), idx);

  % check stop criterions
  STOP1 = isempty(idx); % check regular termination
  STOP2 = fcnEvals > maxFcnEvals;  % check maximal function evaluations
  STOP3 = 5 * length(idx) > maxParts; % check maximal partition

  % regular termination
  if STOP1,
    Q = Q + sum(Q2, 2);
    break;
  end;

  % check if maximal resolution reached
  idxH = find(abs(H) < minH);
  if ~isempty(idxH),
    Q = Q + sum(Q2(idxH), 2);
    idx = setdiff(idx, idxH);
    idxQ = setdiff(idxQ, idxH);
    maxResolutionWarning = true;
    % termination criterion
    if isempty(idx), break, end;
  end;

  % maximal function evaluations reached
  if STOP2,
```

```
  warning('Matlab:adaptiveLobatto:MaxEvaluations',...
    'The maximal number of function evaluations reached; singularity likely.')
  Q = Q + sum(Q2, 2);
  break;
end;

% maximal partition reached
if STOP3,
  warning('Matlab:adaptiveLobatto:parts',...
    'The maximal number of parts reached.')
  Q = Q + sum(Q2, 2);
  break;
end;

% update quadrature value
Q = Q + sum(Q2(:,idxQ) + diffQ(:,idxQ) / 15, 2);

% number of intervals
m = 6 * length(idx);

% initialize t
t = zeros(1, 6 * length(idx));

% hand over new start points A
t(1:6:end) = A(idx); t(2:6:end) = F(idx); t(3:6:end) = D(idx);
t(4:6:end) = C(idx); t(5:6:end) = E(idx); t(6:6:end) = G(idx);
A = t;

% hand over new end points B
t(1:6:end) = F(idx); t(2:6:end) = D(idx); t(3:6:end) = C(idx);
t(4:6:end) = E(idx); t(5:6:end) = G(idx); t(6:6:end) = B(idx);
B = t;

y = zeros(n, 6 * length(idx));
% hand over new start values A
y(:,1:6:end) = yA(:,idx); y(:,2:6:end) = yF(:,idx); y(:,3:6:end) = yD(:,idx);
y(:,4:6:end) = yC(:,idx); y(:,5:6:end) = yE(:,idx); y(:,6:6:end) = yG(:,idx);
yA = y;

% hand over new end values B
y(:,1:6:end) = yF(:,idx); y(:,2:6:end) = yD(:,idx); y(:,3:6:end) = yC(:,idx);
y(:,4:6:end) = yE(:,idx); y(:,5:6:end) = yG(:,idx); y(:,6:6:end) = yB(:,idx);
yB = y;

% widths and midpoints of intervals
H = (B - A) / 2; J = (A + B) / 2;

% calculate new mid points
F = -alpha * H + J; D = -beta * H + J; C =  J;
E =  beta * H + J; G =  alpha * H + J;

% function evaluations
y = fcn([F, D, C, E, G]); fcnEvals = fcnEvals + 5 * m;

% poles at new points
if ~isempty(find(~isfinite(max(abs(y))))), poleWarning = 1; end
```

```
  % hand over new midpoint values of F D C E and G
  yF = y(:,     1 :   m); yD = y(:,   m+1 : 2*m); yC = y(:, 2*m+1 : 3*m);
  yE = y(:, 3*m+1 : 4*m); yG = y(:, 4*m+1 : 5*m);

end;

% display warnings
if ~isfinite(Q),
  warning('Matlab:adaptiveLobatto:Infinite',...
    'The Quadrature of the function reached infinity or is Not-a-Number.')
end;
if maxResolutionWarning,
  warning('Matlab:adaptiveLobatto:MaxResolution',...
    'The maximal resolution of partial interval reached; singularity likely.')
end;
if poleWarning,
  warning('Matlab:adaptiveLobatto:PoleDetection',...
        'A detection of a pole; singularity likely.')
end;

return


% substitute function interval [0, inf] on [0, 1]
function f = infiniteLeft(t, fcn)
f = fcn(log(t)) ;
f = f ./ kron(ones(size(f,1),1), t);
return;

% substitute function interval [-inf, 0] on [0, 1]
function f = infiniteRight(t, fcn)
f = fcn(-log(t));
f = f ./ kron(ones(size(f,1),1), t);
return;

% substitute function interval [-inf, inf] on [-pi / 2, pi / 2]
function f = infiniteBoth(t, fcn)
f = fcn(tan(t));
f = f ./ kron(ones(size(f,1),1), cos(t).^2);
return;
```

References

[dB71] Carl de Boor. *Mathematical Software*, chapter On writing an automatic integration algorithm., pages 201–209. Academic Press, New York, 1971.

[DR84] Philip J. Davis and Philip Rabinowitz. *Methods of numerical integration*. Academic Press, 2 edition, 1984.

[Eie89] Martin C. Eiermann. Automatic, guarenteed integration of analytic functions. *BIT*, 29:270–282, 1989.

[Esp07] Terje O. Espelid. Algorithm 868: Globally doubly adaptive quadrature – reliable matlab codes. *ACM Trans. Math. Softw.*, 33(3):1–21, 2007.

[GG00] Walter Gander and Walter Gautschi. Adaptive quadrature - revisited. Technical Report 306, Eidgenössische Technische Hochschule Zürich, Department Informatik, Institut für Wissenschaftliches Rechnen, August 2000.

[Kah71] D. K. Kahaner. Comparison of numerical quadrature formulas. in Mathematical Software, John R. Rice ed., Academic Press, New York, 1971.

[Kah80] W. Kahan. Handheld calculator evaluates integrals. *Hewlett-Packard Journal*, 31(,8):23–32, 1980.

[KU94] A. R. Krommer and C. W. Ueberhuber. *Numerical Integration on Advanced Computer Systems (Lecture Notes in Computer Science)*. Springer, 1994.

[Pet07] Knut Petras. Principles of verified numerical integration. *Journal of Computational and Applied Mathematics*, 199:317–328, 2007.

[Sha06] L. F. Shampine. Vectorized adaptive quadrature in MATLAB. Preprint submitted to Elsevier Science, 30 October 2006, 2006.