```c
// NumDifInt.c (c) June 5, 2024 by James Pate Williams, Jr.
// Exmaples and Exercises from Chapter 7 of "Numerical
// Analysis: An Algorithmic Approach" (c) 1980 S. D. Conte and Carl
// de Boor Chapter 7 Differentiation and Integration
// Also, we have used code from "A Numerical Library in C
// for Scientists and Engineers" (c) 1994 by H. T. Lau, PhD

#include <direct.h>
#include <float.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const char out_filename[22] = "\\out_NumDifInt.txt";

double f1(double x)
{
    return exp(x);
}

double f2(double x)
{
    return exp(-x);
}

double fDfDx(
    double a,
    double h,
    double (*fx)(double))
{
    // forward difference Equation (7.8)
    return (fx(a + h) - fx(a)) / h;
}

double cDfDx(
    double a,
    double h,
    double (*fx)(double))
{
    // central difference Equation (7.8)
    return (fx(a + h) - fx(a - h)) / (2 * h);
}

double iDfDx(
    double a,
    double h,
    double (*fx)(double))
{
    // interpolation difference Equation (7.10)
    return (-3.0 * fx(a) + 4.0 * fx(a + h)
        - fx(a - 2.0 * h)) / (2 * h);
```

```
 53  }
 54
 55  double cD2fDx2(
 56      double a,
 57      double h,
 58      double (*fx)(double))
 59  {
 60      // central difference Equation (7.14)
 61      return (fx(a - h) - 2 * fx(a) + fx(a + h)) / (h * h);
 62  }
 63
 64  long double Interpolation(
 65      long double x0,
 66      long double x1,
 67      long double (*f)(long double))
 68  {
 69      return (f(x0) - f(x1)) / (x0 - x1);
 70  }
 71
 72  long double dlogxdxl(double x)
 73  {
 74      return 1.0 / x;
 75  }
 76
 77  void Exercise7_1_5(FILE* out_file, int nmax)
 78  {
 79      long double p = 0.0, x0 = 0.0, x1 = 0.0;
 80      long double fxp = dlogxdxl(2.0L);
 81      long double an = 0.0L;
 82      int n;
 83
 84      for (n = 1; n <= nmax; n++)
 85      {
 86          p = pow(2.0, -n);
 87          x0 = 2.0 - p;
 88          x1 = 2.0 + p;
 89          an = Interpolation(x0, x1, logl);
 90      }
 91
 92      fprintf_s(out_file, "f(x) = ln(x)\n");
 93      fprintf_s(out_file, "f'(2)  = %12.10lf\n", fxp);
 94      fprintf_s(out_file, "a[%2ld]  = %12.10Lf\n\n", nmax, an);
 95  }
 96
 97  double RectangleRule(
 98      double a,
 99      double b,
100      double (*f)(double))
101  {
102      return (b - a) * f(a);
103  }
104
```

```c
double MidPointRule(
    double a,
    double b,
    double (*f)(double))
{
    return (b - a) * f((a + b) / 2.0);
}

double TrapezoidalRule(
    double a,
    double b,
    double (*f)(double))
{
    return 0.5 * (b - a) * (f(a) + f(b));
}

double SimpsonsRule(
    double a,
    double b,
    double (*f)(double))
{
    return (b - a) /6.0 * (f(a) + 4.0 * f(0.5 * (a + b)) + f(b));
}

double CorrectedTrapezoidalRule(
    double a,
    double b,
    double (*f)(double),
    double (*g)(double))
{
    double delta = b - a;

    return  (delta / 2.0) * (f(a) + f(b)) +
        (delta * delta / 12.0) * (g(a) - g(b));
}

double NegSqrExp(double x)
{
    return exp(-x * x);
}

double dExp2dx(double x)
{
    return -2.0 * x * exp(-x * x);
}


// Begin functions from "A Numerical Library in C for"
// Scientists and Engineers" (c) 1994 by H. T. Lau, PhD

void system_error(char error_message[])
{
```

```
157     printf("%s", error_message);
158     exit(-1);
159 }
160
161 double** allocate_real_matrix(int lr, int ur, int lc, int uc)
162 {
163     /*  Allocates a real matrix of range [lr..ur][lc..uc].  */
164
165     void system_error(char*);
166     int i;
167     double** p;
168
169     p = (double**)malloc((unsigned)(ur - lr + 1) * sizeof(double*));
170     if (!p) system_error("Failure in allocate_real_matrix().");
171     p -= lr;
172
173     for (i = lr; i <= ur; i++) {
174         p[i] = (double*)malloc((unsigned)(uc - lc + 1) * sizeof(double));
175         if (!p[i]) system_error("Failure in allocate_real_matrix().");
176         p[i] -= lc;
177     }
178     return p;
179 }
180
181 double* allocate_real_vector(int l, int u)
182 {
183     /*  Allocates a real vector of range [l..u].  */
184
185     double* p;
186
187     p = (double*)malloc((unsigned)(u - l + 1) * sizeof(double));
188     if (!p) system_error("Failure in allocate_real_vector().");
189     return p - l;
190 }
191
192 void free_real_matrix(double** m, int lr, int ur, int lc)
193 {
194     /*  Frees a real matrix of range [lr..ur][lc..uc].  */
195
196     int i;
197
198     for (i = ur; i >= lr; i--) free((char*)(m[i] + lc));
199     free((char*)(m + lr));
200 }
201
202 void free_real_vector(double* v, int l)
203 {
204     /*  Frees a real vector of range [l..u].  */
205
206     free((char*)(v + l));
207 }
208
```

```
209 double opfx(
210     char name[],
211     double alpha,
212     double beta,
213     double x,
214     int n)
215 {
216     double a1n = 1, a2n = 1, a3n = 1, a4n = 1;
217     double f0 = 1, f1 = x, f2 = 1;
218 
219     if (strcmp(name, "Laguerre") == 0)
220         f1 = -x + 1;
221 
222     else if (strcmp(name, "Gegenbauer") == 0)
223     {
224         if (alpha != 0)
225             f1 = 2 * alpha * x;
226     }
227 
228     else if (strcmp(name, "Generalized Laguerre") == 0)
229         f1 = alpha + 1 - x;
230 
231     else if (strcmp(name, "Jacobi") == 0)
232         f1 = 0.5 * (alpha - beta + (alpha + beta + 2) * x);
233 
234     for (int k = 2; k <= n; k++)
235     {
236         if (strcmp(name, "Chebyshev") == 0)
237         {
238             a1n = 1.0;
239             a2n = 0.0;
240             a3n = 2.0;
241             a4n = 1.0;
242         }
243 
244         else if (strcmp(name, "Laguerre") == 0)
245         {
246             a1n = k - 1.0 + 1.0;
247             a2n = 2 * (k - 1.0) + 1.0;
248             a3n = -1.0;
249             a4n = k - 1.0;
250         }
251 
252         else if (strcmp(name, "Legendre") == 0)
253         {
254             a1n = k + 1.0 - 1.0;
255             a2n = 0.0;
256             a3n = 2.0 * (k - 1.0) + 1.0;
257             a4n = k - 1.0;
258         }
259 
260         else if (strcmp(name, "Gegenbauer") == 0)
```

```
261         {
262             if (alpha != 0)
263             {
264                 a1n = k - 1.0 + 2.0;
265                 a2n = 0;
266                 a3n = 2 * (k - 1.0 + alpha);
267                 a4n = k - 1.0 + 2.0 * alpha - 1.0;
268             }
269 
270             else
271             {
272                 a1n = 1.0;
273                 a2n = 0.0;
274                 a3n = 2.0;
275                 a4n = 1.0;
276             }
277         }
278 
279         else if (strcmp(name, "Generalized Laguerre") == 0)
280         {
281             a1n = k - 1.0 + 1.0;
282             a2n = 2 * (k - 1.0) + alpha + 1.0;
283             a3n = -1;
284             a4n = k - 1.0 + alpha;
285         }
286 
287         else if (strcmp(name, "Jacobi") == 0)
288         {
289             double t = 2.0 * (k - 1.0) + alpha + beta;
290 
291             a1n = 2.0 * k * (k + alpha + beta) * t;
292             a2n = alpha * alpha - beta * beta;
293             a3n = (t + 1.0) * t * (t + 2.0);
294             a4n = 2 * (k - 1.0 + alpha) * (k - 1.0 + beta) * (t + 2.0);
295         }
296 
297         f2 = ((a2n + a3n * x) * f1 - a4n * f0) / a1n;
298         f0 = f1;
299         f1 = f2;
300     }
301 
302     if (alpha == 0 && strcmp(name, "Gegenbauer") == 0)
303         f2 = 2 * f2 / n;
304 
305     return f2;
306 }
307 
308 double opgx(char name[], double x, int n)
309 {
310     double f0 = 1, f1 = x, f2 = 1;
311 
312     if (strcmp(name, "Laguerre") != 0)
```

```
313     {
314         if (x == 1.0)
315             x -= 0.0000001;
316 
317         else if (x == -1.0)
318             x += 0.0000001;
319     }
320 
321     double g0x = 1.0, g1x = 1.0, g2x = 1.0;
322 
323     if (strcmp(name, "Chebyshev") == 0)
324     {
325         g0x = n;
326         g1x = -n * x;
327         g2x = 1.0 - x * x;
328     }
329 
330     else if (strcmp(name, "Laguerre") == 0)
331     {
332         g0x = -n;
333         g1x = n;
334         g2x = x;
335     }
336 
337     else if (strcmp(name, "Legendre") == 0)
338     {
339         g0x = n;
340         g1x = -n * x;
341         g2x = 1.0 - x * x;
342     }
343 
344     f0 = opfx(name, 0.0, 0.0, x, n - 1);
345     f1 = opfx(name, 0.0, 0.0, x, n);
346     f2 = (g1x * f1 + g0x * f0) / g2x;
347 
348     return f2;
349 }
350 
351 double wf(
352     char name[],
353     double alpha,
354     double beta,
355     double x,
356     int n)
357 {
358     double wx = 0.0;
359 
360     if (strcmp(name, "Gegenbauer") == 0)
361         wx = pow(1 - x * x, alpha - 0.5);
362 
363     else if (strcmp(name, "Generalized Laguerre") == 0)
364         wx = exp(-x) * pow(x, alpha);
```

```
365
366     else if (strcmp(name, "Jacobi") == 0)
367         wx = pow(1 - x, alpha) * pow(1 + x, beta);
368
369     return wx * opfx(name, alpha, beta, x, n);
370 }
371
372 void dupvec(int l, int u, int shift, double a[], double b[])
373 {
374     for (; l <= u; l++) a[l] = b[l + shift];
375 }
376
377 void rotcol(int l, int u, int i, int j, double** a, double c, double s)
378 {
379     double x, y;
380
381     for (; l <= u; l++)
382     {
383         x = a[l][i];
384         y = a[l][j];
385         a[l][i] = x * c + y * s;
386         a[l][j] = y * c - x * s;
387     }
388 }
389
390 int qrivalsymtri(double d[], double bb[], int n, double em[])
391 {
392     int i, i1, low, oldlow, n1, count, max;
393     double bbtol, bbmax, bbi, bbn1, machtol, dn, delta, f, num, shift, g, h,
394         t, p, r, s, c, oldg;
395
396     t = em[2] * em[1];
397     bbtol = t * t;
398     machtol = em[0] * em[1];
399     max = (int)em[4];
400     bbmax = 0.0;
401     count = 0;
402     oldlow = n;
403     n1 = n - 1;
404     while (n > 0)
405     {
406         i = n;
407         do
408         {
409             low = i;
410             i--;
411         } while ((i >= 1) ? (bb[i] > bbtol ? 1 : 0) : 0);
412         if (low > 1)
413             if (bb[low - 1] > bbmax) bbmax = bb[low - 1];
414         if (low == n)
415             n = n1;
416         else
```

```
417         {
418             dn = d[n];
419             delta = d[n1] - dn;
420             bbn1 = bb[n1];
421             if (fabs(delta) < machtol)
422                 r = sqrt(bbn1);
423             else
424             {
425                 f = 2.0 / delta;
426                 num = bbn1 * f;
427                 r = -num / (sqrt(num * f + 1.0) + 1.0);
428             }
429             if (low == n1)
430             {
431                 d[n] = dn + r;
432                 d[n1] -= r;
433                 n -= 2;
434             }
435             else
436             {
437                 count++;
438                 if (count > max) break;
439                 if (low < oldlow)
440                 {
441                     shift = 0.0;
442                     oldlow = low;
443                 }
444                 else
445                     shift = dn + r;
446                 h = d[low] - shift;
447                 if (fabs(h) < machtol)
448                     h = (h <= 0.0) ? -machtol : machtol;
449                 g = h;
450                 t = g * h;
451                 bbi = bb[low];
452                 p = t + bbi;
453                 i1 = low;
454                 for (i = low + 1; i <= n; i++)
455                 {
456                     s = bbi / p;
457                     c = t / p;
458                     h = d[i] - shift - bbi / h;
459                     if (fabs(h) < machtol)
460                         h = (h <= 0.0) ? -machtol : machtol;
461                     oldg = g;
462                     g = h * c;
463                     t = g * h;
464                     d[i1] = oldg - g + d[i];
465                     bbi = (i == n) ? 0.0 : bb[i];
466                     p = t + bbi;
467                     bb[i1] = s * p;
468                     i1 = i;
```

```
469                 }
470                 d[n] = g + shift;
471             }
472         }
473         n1 = n - 1;
474     }
475     em[3] = sqrt(bbmax);
476     em[5] = count;
477     return n;
478 }
479
480 void allzerortpol(int n, double b[], double c[],
481     double zer[], double em[])
482 {
483     int i;
484     double nrm;
485     double* bb = allocate_real_vector(0, n + 1);
486
487     nrm = fabs(b[0]);
488     for (i = 1; i <= n - 2; i++)
489         if (c[i] + fabs(b[i]) > nrm) nrm = c[i] + fabs(b[i]);
490     if (n > 1)
491         nrm = (nrm + 1 >= c[n - 1] + fabs(b[n - 1])) ? nrm + 1.0 :
492         (c[n - 1] + fabs(b[n - 1]));
493     em[1] = nrm;
494     for (i = n; i >= 1; i--) zer[i] = b[i - 1];
495     dupvec(1, n - 1, 0, bb, c);
496     qrivalsymtri(zer, bb, n, em);
497     free_real_vector(bb, 0);
498 }
499
500 void alljaczer(int n, double alfa, double beta, double zer[])
501 {
502     double sum, min, gamma, zeri;
503     double* a = NULL, *b = NULL;
504     double em[6] = { 0 };
505     int i, m;
506
507     if (alfa == beta)
508     {
509         a = allocate_real_vector(0, n / 2 + 1);
510         b = allocate_real_vector(0, n / 2 + 1);
511         m = n / 2;
512         if (n != 2 * m)
513         {
514             gamma = 0.5;
515             zer[m + 1] = 0.0;
516         }
517         else
518             gamma = -0.5;
519         min = 0.25 - alfa * alfa;
520         sum = alfa + gamma + 2.0;
```

```
521         a[0] = (gamma - alfa) / sum;
522         a[1] = min / sum / (sum + 2.0);
523         b[1] = 4.0 * (1.0 + alfa) * (1.0 + gamma) / sum / sum / (sum + 1.0);
524         for (i = 2; i <= m - 1; i++)
525         {
526             sum = (double)i + i + alfa + gamma;
527             a[i] = min / sum / (sum + 2.0);
528             sum *= sum;
529             b[i] = 4.0 * i * (i + alfa + gamma) * (i + alfa) * (i + gamma) /
                  sum / (sum - 1.0);
530         }
531         em[0] = DBL_MIN;
532         em[2] = DBL_EPSILON;
533         em[4] = 6.0 * m;
534         allzerortpol(m, a, b, zer, em);
535         for (i = 1; i <= m; i++)
536         {
537             zer[i] = zeri = -sqrt((1.0 + zer[i]) / 2.0);
538             zer[n + 1 - i] = -zeri;
539         }
540     }
541     else
542     {
543         a = allocate_real_vector(0, n + 1);
544         b = allocate_real_vector(0, n + 1);
545         min = (beta - alfa) * (beta + alfa);
546         sum = alfa + beta + 2.0;
547         b[0] = 0.0;
548         a[0] = (beta - alfa) / sum;
549         a[1] = min / sum / (sum + 2.0);
550         b[1] = 4.0 * (1.0 + alfa) * (1.0 + beta) / sum / sum / (sum + 1.0);
551         for (i = 2; i <= n - 1; i++)
552         {
553             sum = (double)i + i + alfa + beta;
554             a[i] = min / sum / (sum + 2.0);
555             sum *= sum;
556             b[i] = 4.0 * i * (i + alfa + beta) * (i + alfa) * (i + beta) / (sum
                  - 1.0) / sum;
557         }
558         em[0] = DBL_MIN;
559         em[2] = DBL_EPSILON;
560         em[4] = 6.0 * n;
561         allzerortpol(n, a, b, zer, em);
562     }
563
564     free_real_vector(a, 0);
565     free_real_vector(b, 0);
566 }
567
568 void alllagzer(int n, double alfa, double zer[])
569 {
570     double* a = NULL, *b = NULL, em[6] = { 0 };
```

```
571     int i;
572
573     a = allocate_real_vector(0, n + 1);
574     b = allocate_real_vector(0, n + 1);
575     b[0] = 0.0;
576     a[n - 1] = (double)n + n + alfa - 1.0;
577     for (i = 1; i <= n - 1; i++)
578     {
579         a[i - 1] = (double)i + i + alfa - 1.0;
580         b[i] = i * (i + alfa);
581     }
582     em[0] = DBL_MIN;
583     em[2] = DBL_EPSILON;
584     em[4] = 6.0 * n;
585     allzerortpol(n, a, b, zer, em);
586     free_real_vector(a, 0);
587     free_real_vector(b, 0);
588 }
589
590 void Zeros(
591     char name[],
592     double alpha,
593     double beta,
594     int n,
595     double roots[])
596 {
597     double* zer = allocate_real_vector(0, n + 1);
598     int i;
599
600     if (strcmp(name, "Legendre") == 0)
601     {
602         double* b = allocate_real_vector(0, n);
603         double* c = allocate_real_vector(0, n);
604         double em[6] = { 0 };
605
606         for (i = 0; i < n; i++)
607         {
608             b[i] = 0.0;
609             c[i] = (double)i * i / (4.0 * i * i - 1.0);
610         }
611
612         em[0] = em[2] = 1.0E-10;
613         em[4] = 5.0 * n;
614
615         allzerortpol(n, b, c, zer, em);
616     }
617
618     else if (strcmp(name, "Gegenbauer") == 0)
619     {
620         if (alpha > 0)
621             alljaczer(n, alpha - 0.5, alpha - 0.5, zer);
622
```

```
623         else if (alpha == 0)
624         {
625             double* b = allocate_real_vector(0, n);
626             double* c = allocate_real_vector(0, n);
627             double em[6] = { 0 };
628
629             c[1] = 0.5;
630
631             for (int k = 2; k < n; k++)
632                 c[k] = 0.25;
633
634             em[0] = em[2] = 1.0E-10;
635             em[4] = 5.0 * n;
636
637             allzerortpol(n, b, c, zer, em);
638             free_real_vector(b, 0);
639             free_real_vector(c, 0);
640         }
641     }
642
643     else if (strcmp(name, "Generalized Laguerre") == 0)
644         alllagzer(n, alpha, zer);
645
646     else if (strcmp(name, "Jacobi") == 0)
647         alljaczer(n, alpha, beta, zer);
648
649     for (i = 1; i <= n; i++)
650         roots[i] = zer[i];
651
652     free_real_vector(zer, 0);
653 }
654
655 // end H. T. Lau, PhD code
656
657 double ExtTrapezoidalRule(int n, double a, double b, double (*f)(double))
658 {
659     double h = (b - a) / n;
660     double s = 0.5 * (f(a) + f(b));
661     double x = a + h;
662
663     for (int i = 1; i < n; i++)
664     {
665         s += f(x);
666         x += h;
667     }
668
669     return h * s;
670 }
671
672 double ExtCorrTrapezoidalRule(
673     int n, double a, double b,
674     double (*f)(double),
```

```
675      double (*dfdx)(double))
676  {
677      double h = (b - a) / n;
678      double s = 0.5 * (f(a) + f(b));
679      double x = a + h;
680
681      for (int i = 1; i < n; i++)
682      {
683          s += f(x);
684          x += h;
685      }
686
687      s = h * s + h * h * (dfdx(a) - dfdx(b)) / 12.0;
688      return s;
689  }
690
691  double ExtSimpsonsRule(int n, double a, double b, double (*f)(double))
692  {
693      double h = (b - a) / n;
694      double h2 = 2.0 * h;
695      double s = 0.0;
696      double t = 0.0;
697      double x = a + h;
698
699      for (int i = 1; i < n; i += 2)
700      {
701          s += f(x);
702          x += h2;
703      }
704
705      x = a + h2;
706
707      for (int i = 2; i < n; i += 2)
708      {
709          t += f(x);
710          x += h2;
711      }
712
713      return h * (f(a) + 4 * s + 2 * t + f(b)) / 3.0;
714  }
715
716  double AdaptiveExtSimpsonsRule(int n, double a, double b, double (*f)(double))
717  {
718      double h = (b - a) / n;
719      double s = 0.0;
720
721      for (int i = 0; i <= n - 1; i++)
722      {
723          double xi = a + i * h;
724          double x4 = xi + 0.25 * h;
725          double x2 = xi + 0.50 * h;
726          double x3 = xi + 0.75 * h;
```

```
727         double x1 = a + (i + 1.0) * h;
728
729         s += f(xi) + 4.0 * f(x4) + 2.0 * f(x2) +
730             4.0 * f(x3) + f(x1);
731     }
732
733     return s * h / 12.0;
734 }
735
736 void GenerateGaussianLegendreAbscissasAndWeights(int n, double x[], double w[])
737 {
738     double* roots = allocate_real_vector(0, n);
739     Zeros("Legendre", 0.0, 0.0, n, roots);
740
741     for (int i = 1; i <= n; i++)
742     {
743         double xi = roots[i];
744         double fd = opgx("Legendre", xi, n);
745         double x2 = 1.0 - xi * xi;
746
747         x[i] = xi;
748         w[i] = 2.0 / (x2 * fd * fd);
749     }
750
751     free_real_vector(roots, 0);
752 }
753
754 double sinx2divx(double x)
755 {
756     double sinx = sin(x);
757     return sinx * sinx / x;
758 }
759
760 double GLIntegrate11(int n, double x[], double w[], double (*f)(double))
761 {
762     double sum = 0.0;
763
764     for (int i = 1; i <= n; i++)
765         sum += w[i] * f(x[i]);
766
767     return sum;
768 }
769
770 double GLIntegrateAB(int n, double a, double b,
771     double x[], double w[], double (*f)(double))
772 {
773     double c = (b - a) / 2.0;
774     double d = (b + a) / 2.0;
775     double sum = 0.0;
776
777     for (int i = 1; i <= n; i++)
778         sum += w[i] * f(c * x[i] + d);
```

```
779
780     return c * sum;
781 }
782
783 void CreateTable7_1(FILE* out_file)
784 {
785     double h[] = { 1.0, 0.1, 0.01, 0.001, 0.0001 };
786     int i = 0;
787
788     fprintf_s(out_file, "From Numerical Analysis: ");
789     fprintf_s(out_file, "An Algorithmic Approach\n");
790     fprintf_s(out_file, "(c) 1980 S. D. Conte and ");
791     fprintf_s(out_file, "Carl de Boor\n\n");
792     fprintf_s(out_file, "Table 7.1\n\n");
793     fprintf_s(out_file, "  h\t       exp(h)\t\t      exp(-h)\t\t");
794     fprintf_s(out_file, "        Dh\t\t       D2h\n");
795
796     for (i = 0; i < 5; i++)
797     {
798         double hi = h[i], x = 0;
799         double Dh = cDfDx(x, hi, f1);
800         double Dh2 = cD2fDx2(x, hi, f2);
801
802         fprintf_s(out_file, "%6.4lf\t", hi);
803         fprintf_s(out_file, "%17.10e\t%17.10e\t", f1(hi), f2(hi));
804         fprintf_s(out_file, "%17.10e\t%17.10e\n", Dh, Dh2);
805     }
806 }
807
808 void Exercise7_1_1(FILE* out_file)
809 {
810     double h = 0.1;
811     double x[] = { 1.2 , 1.3, 1.4, 1.5, 1.6 };
812     double f[] = { 1.5095, 1.6984, 1.9043,
813         2.1293, 2.3756 };
814     double fdfdx = (f[3] - f[2]) / h;
815     double cdfdx = (f[3] - f[1]) / (2.0 * h);
816     double idfdx = (-3.0 * f[2] + 4.0 * f[3] - f[4]) / (2.0 * h);
817     double cd2fdx2 = (f[1] - 2.0 * f[2] + f[3]) / (h * h);
818     double fp = cosh(1.4), fpp = sinh(1.4);
819     double error1 = fabs(fp - fdfdx);
820     double error2 = fabs(fp - cdfdx);
821     double error3 = fabs(fp - idfdx);
822     double error4 = fabs(fpp - cd2fdx2);
823
824     fprintf_s(out_file, "Exercise 7.1-1\n\n");
825
826     fprintf_s(out_file, "%s", "f'(1.4) = cosh(1.4) = ");
827     fprintf_s(out_file, "%8.6lf\n", fp);
828     fprintf_s(out_file, "%s", "Forward f'(1.4)     = ");
829     fprintf_s(out_file, "%8.6lf\n", fdfdx);
830     fprintf_s(out_file, "%s", "Error               = ");
```

```
831     fprintf_s(out_file, "%8.6lf\n", error1);
832     fprintf_s(out_file, "\n");
833
834     fprintf_s(out_file, "%s", "Central f'(1.4)     = ");
835     fprintf_s(out_file, "%8.6lf\n", cdfdx);
836     fprintf_s(out_file, "%s", "Error               = ");
837     fprintf_s(out_file, "%8.6lf", error2);
838     fprintf_s(out_file, "\n\n");
839
840     fprintf_s(out_file, "%s", "Interpo f'(1.4)     = ");
841     fprintf_s(out_file, "%8.6lf\n", idfdx);
842     fprintf_s(out_file, "%s", "Error               = ");
843     fprintf_s(out_file, "%8.6lf", error3);
844     fprintf_s(out_file, "\n\n");
845
846     fprintf_s(out_file, "%s", "f''(1.4) = sinh(1.4) = ");
847     fprintf_s(out_file, "%8.6lf\n", fpp);
848     fprintf_s(out_file, "%s", "Central f''(1.4)     = ");
849     fprintf_s(out_file, "%8.6lf\n", cd2fdx2);
850     fprintf_s(out_file, "%s", "Error                = ");
851     fprintf_s(out_file, "%8.6lf", error4);
852     fprintf_s(out_file, "\n\n");
853 }
854
855 void Exercise7_1_2(FILE* out_file)
856 {
857     double x[] = { 0.398, 0.399, 0.400, 0.401, 0.402 };
858     double f[] = { 0.408591, 0.409671, 0.410752,
859         0.411834, 0.412915 };
860     double h1 = 0.001, h2 = 0.002;
861     double cdfdx1 = (f[3] - f[1]) / (2.0 * h1);
862     double cdfdx2 = (f[4] - f[0]) / (2.0 * h2);
863     double fp = cosh(0.4);
864     double error1 = fabs(fp - cdfdx1);
865     double error2 = fabs(fp - cdfdx2);
866
867     fprintf_s(out_file, "Exercise 7.1-2\n\n");
868
869     fprintf_s(out_file, "% s", "f'(0.4) = cosh(0.4) = ");
870     fprintf_s(out_file, "%12.10lf\n", cosh(0.4));
871     fprintf_s(out_file, "%s", "Central1 f'(0.4)    = ");
872     fprintf_s(out_file, "%12.10lf\n", cdfdx1);
873     fprintf_s(out_file, "%s", "Error               = ");
874     fprintf_s(out_file, "%12.10lf", error1);
875     fprintf_s(out_file, "\n\n");
876
877     fprintf_s(out_file, "% s", "f'(0.4) = cosh(0.4) = ");
878     fprintf_s(out_file, "%12.10lf\n", cosh(0.4));
879     fprintf_s(out_file, "%s", "Central2 f'(0.4)    = ");
880     fprintf_s(out_file, "%12.10lf\n", cdfdx2);
881     fprintf_s(out_file, "%s", "Error               = ");
882     fprintf_s(out_file, "%12.10lf", error2);
```

```
883     fprintf_s(out_file, "\n\n");
884 }
885
886 void Exercise7_1_3(FILE* out_file)
887 {
888     double h = 0.1, x = 0, fp = cosh(x);
889     double E_plus = 0.0, E_minus = 0.0;
890     double errorH = 0.0;
891
892     do
893     {
894         double y1 = x - h, y2 = x + h;
895         double fy1 = sinh(x + h);
896         double fy2 = sinh(x - h);
897         double fp1 = (fy1 - fy2) / (2.0 * h);
898         errorH = fabs(fy1 - fy2);
899         h /= 1.0000005;
900     } while (errorH > 0.5e-7);
901
902     fprintf_s(out_file, "Exercise 7.1-3\n\n");
903     fprintf_s(out_file, "E_+ - E_- = %16.10e\n", errorH);
904     fprintf_s(out_file, "h         = %16.10e\n\n", h);
905 }
906
907 void Example7_2(FILE* out_file)
908 {
909     double a = 0, b = 1, exact = 0.74682;
910     double R = RectangleRule(a, b, NegSqrExp);
911     double M = MidPointRule(a, b, NegSqrExp);
912     double T = TrapezoidalRule(a, b, NegSqrExp);
913     double S = SimpsonsRule(a, b, NegSqrExp);
914     double C = CorrectedTrapezoidalRule(
915         a, b, NegSqrExp, dExp2dx);
916     double ER = fabs(R - exact);
917     double EM = fabs(M - exact);
918     double ET = fabs(T - exact);
919     double ES = fabs(S - exact);
920     double EC = fabs(C - exact);
921
922     fprintf_s(out_file, "Example 7.2.1\n\n");
923
924     fprintf_s(out_file, "f(x) = exp(-x * x), a = 0, b = 1\n");
925     fprintf_s(out_file, "%s", "Rectangle Rule   = ");
926     fprintf_s(out_file, "%7.5lf\n", R);
927     fprintf_s(out_file, "%s", "Error            = ");
928     fprintf_s(out_file, "%7.5lf\n", ER);
929
930     fprintf_s(out_file, "%s", "Mid-point Rule   = ");
931     fprintf_s(out_file, "%7.5lf\n", M);
932     fprintf_s(out_file, "%s", "Error            = ");
933     fprintf_s(out_file, "%7.5lf\n", EM);
934
```

```
935     fprintf_s(out_file, "%s", "Trapezoidal Rule = ");
936     fprintf_s(out_file, "%7.5lf\n", T);
937     fprintf_s(out_file, "%s", "Error            = ");
938     fprintf_s(out_file, "%7.5lf\n", ET);
939
940     fprintf_s(out_file, "%s", "Simpson's Rule   = ");
941     fprintf_s(out_file, "%7.5lf\n", S);
942     fprintf_s(out_file, "%s", "Error            = ");
943     fprintf_s(out_file, "%7.5lf\n", ES);
944
945     fprintf_s(out_file, "%s", "Corrected Rule   = ");
946     fprintf_s(out_file, "%7.5lf\n", C);
947     fprintf_s(out_file, "%s", "Error            = ");
948     fprintf_s(out_file, "%7.5lf\n\n", EC);
949 }
950
951 void Example7_2a(FILE* out_file)
952 {
953     int n = 4;
954     double a = 0.0, b = 1.0;
955     double error = 0.0, gi = 0.0;
956     double* w = allocate_real_vector(0, n);
957     double* x = allocate_real_vector(0, n);
958     int i;
959
960     GenerateGaussianLegendreAbscissasAndWeights(n, x, w);
961     gi = GLIntegrateAB(n, a, b, x, w, NegSqrExp);
962
963     fprintf_s(out_file, "Example 7.2a\n\n");
964
965     fprintf_s(out_file, "f(x) = exp(-x * x), a = 0, b = 1\n");
966     fprintf_s(out_file, "Abscissa\t\tWeight\n");
967
968     for (i = 1; i <= n; i++)
969         fprintf_s(out_file, "x[%ld] = %+10.8lf\tw[%ld] = %+10.8lf\n",
970             i, x[i], i, w[i]);
971
972     fprintf_s(out_file, "\n");
973     fprintf_s(out_file, "%s", "Gaussian Quadrature = ");
974     fprintf_s(out_file, "%10.8lf\n\n", gi);
975
976     free_real_vector(w, 0);
977     free_real_vector(x, 0);
978 }
979
980 void Example7_3(FILE* out_file)
981 {
982     int n = 4;
983     double a = 1.0, b = 3.0;
984     double error = 0.0, exact = 0.79482518, gi = 0.0;
985     double* w = allocate_real_vector(0, n);
986     double* x = allocate_real_vector(0, n);
```

```
 987      int i;
 988
 989      GenerateGaussianLegendreAbscissasAndWeights(n, x, w);
 990      gi = GLIntegrateAB(n, a, b, x, w, sinx2divx);
 991
 992      fprintf_s(out_file, "Example 7.3\n\n");
 993      fprintf_s(out_file, "Abscissa\t\tWeight\n");
 994
 995      fprintf_s(out_file, "f(x) = (sin(x))^2 / x, a = 1, b = 3\n");
 996      for (i = 1; i <= n; i++)
 997          fprintf_s(out_file, "x[%ld] = %+10.8lf\tw[%ld] = %+10.8lf\n",
 998              i, x[i], i, w[i]);
 999
1000      fprintf_s(out_file, "\n");
1001      fprintf_s(out_file, "%s", "Gaussian Quadrature = ");
1002      fprintf_s(out_file, "%10.8lf\n", gi);
1003      fprintf_s(out_file, "%s", "Error               = ");
1004      fprintf_s(out_file, "%10.8lf\n\n", fabs(gi - exact));
1005
1006      free_real_vector(w, 0);
1007      free_real_vector(x, 0);
1008  }
1009
1010  double xsinx(double x)
1011  {
1012      return x * sin(x);
1013  }
1014
1015  double dxsinxdx(double x)
1016  {
1017      return (sin(x) + x * cos(x));
1018  }
1019
1020  void Exercise7_2_2(FILE* out_file)
1021  {
1022      double a = 0, b = 1, exact = sin(1) - cos(1);
1023      double R = RectangleRule(a, b, xsinx);
1024      double M = MidPointRule(a, b, xsinx);
1025      double T = TrapezoidalRule(a, b, xsinx);
1026      double S = SimpsonsRule(a, b, xsinx);
1027      double C = CorrectedTrapezoidalRule(a, b, xsinx, dxsinxdx);
1028      double ER = fabs(R - exact);
1029      double EM = fabs(M - exact);
1030      double ET = fabs(T - exact);
1031      double ES = fabs(S - exact);
1032      double EC = fabs(C - exact);
1033
1034      fprintf_s(out_file, "Exercise 7.2-2\n\n");
1035
1036      fprintf_s(out_file, "f(x) = x * sin(x), a = 0, b = 1\n");
1037      fprintf_s(out_file, "%s", "Rectangle Rule   = ");
1038      fprintf_s(out_file, "%10.8lf\n", R);
```

```
1039      fprintf_s(out_file, "%s", "Error           = ");
1040      fprintf_s(out_file, "%10.8lf\n", ER);
1041
1042      fprintf_s(out_file, "%s", "Mid-point Rule   = ");
1043      fprintf_s(out_file, "%10.8lf\n", M);
1044      fprintf_s(out_file, "%s", "Error           = ");
1045      fprintf_s(out_file, "%10.8lf\n", EM);
1046
1047      fprintf_s(out_file, "%s", "Trapezoidal Rule = ");
1048      fprintf_s(out_file, "%10.8lf\n", T);
1049      fprintf_s(out_file, "%s", "Error           = ");
1050      fprintf_s(out_file, "%10.8lf\n", ET);
1051
1052      fprintf_s(out_file, "%s", "Simpson's Rule   = ");
1053      fprintf_s(out_file, "%10.8lf\n", S);
1054      fprintf_s(out_file, "%s", "Error           = ");
1055      fprintf_s(out_file, "%10.8lf\n", ES);
1056
1057      fprintf_s(out_file, "%s", "Corrected Rule   = ");
1058      fprintf_s(out_file, "%10.8lf\n", C);
1059      fprintf_s(out_file, "%s", "Error           = ");
1060      fprintf_s(out_file, "%10.8lf\n\n", EC);
1061  }
1062
1063  double F(double x)
1064  {
1065      double result = 0.0;
1066
1067      if (x >= 0 && x <= 0.5)
1068          result = x;
1069      else if (x >= 0.5 && x <= 1.0)
1070          result = 1.0 - x;
1071
1072      return result;
1073  }
1074
1075  double dFdx(double x)
1076  {
1077      double result = 0.0;
1078
1079      if (x >= 0 && x <= 0.5)
1080          result = 1;
1081      else if (x >= 0.5 && x <= 1.0)
1082          result = -1.0;
1083
1084      return result;
1085  }
1086
1087  void Exercise7_2_3(FILE* out_file)
1088  {
1089      double S = SimpsonsRule(0.0, 1.0, F);
1090      double C = CorrectedTrapezoidalRule(0.0, 1.0, F, dFdx);
```

```
1091      double T = TrapezoidalRule(0.0, 1.0, F);
1092
1093      fprintf_s(out_file, "Exercise 7.2-3\n\n");
1094
1095      fprintf_s(out_file, "f(x) = x, a = 0, b = 1 / 2\n");
1096      fprintf_s(out_file, "f(x) = 1 - x, a = 1 / 2, b = 1\n");
1097      fprintf_s(out_file, "%s", "Trapezoidal Rule = ");
1098      fprintf_s(out_file, "%10.8lf\n", T);
1099
1100      T = TrapezoidalRule(0.0, 0.5, F) +
1101          TrapezoidalRule(0.5, 1.0, F);
1102
1103      fprintf_s(out_file, "%s", "Trapezoidal Rule = ");
1104      fprintf_s(out_file, "%10.8lf\n", T);
1105      fprintf_s(out_file, "%s", "Simpson's Rule   = ");
1106      fprintf_s(out_file, "%10.8lf\n", S);
1107      fprintf_s(out_file, "%s", "Corrected Rule   = ");
1108      fprintf_s(out_file, "%10.8lf\n\n", C);
1109  }
1110
1111  double Fx(double x)
1112  {
1113      return pow(1.0 - x * x, 1.5);
1114  }
1115
1116  void Exercise7_2_5(FILE* out_file)
1117  {
1118      double S = SimpsonsRule(0.0, 1.0, Fx);
1119
1120      fprintf_s(out_file, "Exercise 7.2-5\n\n");
1121      fprintf_s(out_file, "%s", "Simpson's Rule   = ");
1122      fprintf_s(out_file, "%10.8lf\n\n", S);
1123  }
1124
1125  void Exercise7_2_6(FILE* out_file)
1126  {
1127      double T = TrapezoidalRule(0.0, 1.0, NegSqrExp);
1128
1129      fprintf_s(out_file, "Exercise 7.2-6\n\n");
1130      fprintf_s(out_file, "f(x) = exp(-x * x), a = 0, b = 1\n");
1131      fprintf_s(out_file, "%s", "Trapezoidal Rule   = ");
1132      fprintf_s(out_file, "%10.8lf\n\n", T);
1133  }
1134
1135  void Exercise7_3_4(FILE* out_file)
1136  {
1137      int n = 6, N = 1024;
1138      double a = 0.0, b = 32.0, pi = 4.0 * atan(1.0);
1139      double error = 0.0, exact = sqrt(pi) / 2.0, gi = 0.0;
1140      double* w = allocate_real_vector(0, n);
1141      double* x = allocate_real_vector(0, n);
1142      double S = ExtSimpsonsRule(N, a, b, NegSqrExp);
```

```
1143      double T = ExtTrapezoidalRule(N, a, b, NegSqrExp);
1144
1145      GenerateGaussianLegendreAbscissasAndWeights(n, x, w);
1146      gi = GLIntegrateAB(n, a, b, x, w, NegSqrExp);
1147
1148      fprintf_s(out_file, "Exercise 7.3-4\n\n");
1149
1150      fprintf_s(out_file, "f(x) = exp(-x * x), a = 0, b = 1\n");
1151      fprintf_s(out_file, "%s", "Gaussian Quadrature  = ");
1152      fprintf_s(out_file, "%10.8lf\n", gi);
1153      fprintf_s(out_file, "%s", "Error                = ");
1154      fprintf_s(out_file, "%10.8lf\n\n", fabs(gi - exact));
1155
1156      fprintf_s(out_file, "%s", "Ext Trapezoidal Rule = ");
1157      fprintf_s(out_file, "%10.8lf\n", T);
1158      fprintf_s(out_file, "%s", "Error                = ");
1159      fprintf_s(out_file, "%10.8lf\n\n", fabs(T - exact));
1160
1161      fprintf_s(out_file, "%s", "Ext Simpson's Rule   = ");
1162      fprintf_s(out_file, "%10.8lf\n", S);
1163      fprintf_s(out_file, "%s", "Error                = ");
1164      fprintf_s(out_file, "%10.8lf\n\n", fabs(S - exact));
1165
1166      free_real_vector(w, 0);
1167      free_real_vector(x, 0);
1168  }
1169
1170  int Nn = 0;
1171
1172  double xn(double x)
1173  {
1174      return pow(x, Nn);
1175  }
1176
1177  void Exercise7_3_9(FILE* out_file)
1178  {
1179      int n = 0;
1180      double a = -1.0, b = 1.0, pi = 4.0 * atan(1.0);
1181      double error = 0.0, exact = sqrt(pi) / 2.0, gi = 0.0;
1182      double* w = allocate_real_vector(0, n);
1183      double* x = allocate_real_vector(0, n);
1184
1185      fprintf_s(out_file, "Exercise 7.3-9\n");
1186      fprintf_s(out_file, "f(x) = [1 - pow(-1, n + 1)] / (n + 1)\n\n");
1187
1188      for (n = 0; n <= 8; n++)
1189      {
1190          Nn = n;
1191          double exact = (1 - pow(-1.0, n + 1.0)) / (n + 1.0);
1192          double* w = allocate_real_vector(0, n);
1193          double* x = allocate_real_vector(0, n);
1194
```

```
1195            GenerateGaussianLegendreAbscissasAndWeights(n, x, w);
1196            gi = GLIntegrate11(n, x, w, xn);
1197
1198            fprintf_s(out_file, "f(x) = x^n, a = -1, b = 1\n");
1199            fprintf_s(out_file, "n = %ld\n", n);
1200            fprintf_s(out_file, "%s", "Gaussian Quadrature  = ");
1201            fprintf_s(out_file, "%+10.8lf\n", gi);
1202            fprintf_s(out_file, "%s", "Error                = ");
1203            fprintf_s(out_file, "%+10.8lf\n", fabs(gi - exact));
1204
1205            free_real_vector(w, 0);
1206            free_real_vector(x, 0);
1207        }
1208
1209        fprintf_s(out_file, "\n");
1210    }
1211
1212    double xlnx(double x)
1213    {
1214        return x * log(x);
1215    }
1216
1217    void Exercise7_4_3(FILE* out_file)
1218    {
1219        int N = 0;
1220        double a = 1.0, b = 2.0;
1221
1222        fprintf_s(out_file, "Exercise 7.3-9\n\n");
1223
1224        for (N = 10; N <= 20; N += 10)
1225        {
1226            double S = ExtSimpsonsRule(N, a, b, xlnx);
1227
1228            fprintf_s(out_file, "f(x) = x * ln(x) a = 1, b = 2\n");
1229            fprintf_s(out_file, "%s", "Ext Simpson's Rule = ");
1230            fprintf_s(out_file, "%10.8lf\n\n", S);
1231        }
1232    }
1233
1234    double xexpx(double x)
1235    {
1236        return x * exp(-x);
1237    }
1238
1239    double xcosx(double x)
1240    {
1241        return x * cos(x);
1242    }
1243
1244    double powx2(double x)
1245    {
1246        return pow(1.0 + x * x, 1.5);
```

```
1247  }
1248
1249  void Exercise7_4_4(FILE* out_file)
1250  {
1251      int N = 0;
1252      double a = 1.0, b = 2.0, d = 0.0, integ1 = 0.0, integ2 = 0.0;
1253
1254      fprintf_s(out_file, "Exercise 7.4-4\n\n");
1255      fprintf_s(out_file, "f(x) = x * exp(-x) a = 0, b = 1\n");
1256
1257      for (N = 10; N <= 100; N *= 2)
1258      {
1259          double S = ExtSimpsonsRule(N, a, b, xexpx);
1260
1261          if (N == 10)
1262              integ1 = S;
1263          else
1264          {
1265              integ2 = S;
1266              d = fabs(integ1 - integ2);
1267              integ1 = integ2;
1268          }
1269
1270          fprintf_s(out_file, "%s", "Ext Simpson's Rule = ");
1271          fprintf_s(out_file, "%10.8lf\t", S);
1272          fprintf_s(out_file, "%s", "Delta = ");
1273          fprintf_s(out_file, "%10.8lf\t", d);
1274          fprintf_s(out_file, "%s", "N = ");
1275          fprintf_s(out_file, "%ld\n", N);
1276
1277          if (N > 10 && d < 1.0e-6)
1278              break;
1279      }
1280
1281      d = 0.0;
1282      fprintf_s(out_file, "f(x) = x * cos(x) a = 0, b = 1\n");
1283
1284      for (N = 10; N <= 100; N *= 2)
1285      {
1286          double S = ExtSimpsonsRule(N, a, b, xcosx);
1287
1288          if (N == 10)
1289              integ1 = S;
1290          else
1291          {
1292              integ2 = S;
1293              d = fabs(integ1 - integ2);
1294
1295              integ1 = integ2;
1296          }
1297
1298          fprintf_s(out_file, "%s", "Ext Simpson's Rule = ");
```

```
1299            fprintf_s(out_file, "%10.8lf\t", S);
1300            fprintf_s(out_file, "%s", "Delta = ");
1301            fprintf_s(out_file, "%10.8lf\t", d);
1302            fprintf_s(out_file, "%s", "N = ");
1303            fprintf_s(out_file, "%ld\n", N);
1304
1305            if (N > 10 && d < 1.0e-6)
1306                break;
1307        }
1308
1309        d = 0.0;
1310        fprintf_s(out_file, "f(x) = (1 + x * x) ^ 1.5 a = 0, b = 1\n");
1311
1312        for (N = 10; N <= 100; N *= 2)
1313        {
1314            double S = ExtSimpsonsRule(N, a, b, powx2);
1315
1316            if (N == 10)
1317                integ1 = S;
1318            else
1319            {
1320                integ2 = S;
1321                d = fabs(integ1 - integ2);
1322                integ1 = integ2;
1323            }
1324
1325            fprintf_s(out_file, "%s", "Ext Simpson's Rule = ");
1326            fprintf_s(out_file, "%10.8lf\t", S);
1327            fprintf_s(out_file, "%s", "Delta = ");
1328            fprintf_s(out_file, "%10.8lf\t", d);
1329            fprintf_s(out_file, "%s", "N = ");
1330            fprintf_s(out_file, "%ld\n", N);
1331
1332            if (N > 10 && d < 1.0e-6)
1333                break;
1334        }
1335
1336        fprintf_s(out_file, "\n");
1337    }
1338
1339    double dxlnxdx(double x)
1340    {
1341        return (log(x) + 1.0);
1342    }
1343
1344    void Exercise7_4_5(FILE* out_file)
1345    {
1346        double a = 1.0, b = 2.0;
1347        double d = 0.0, integ1 = 0.0, integ2 = 0.0;
1348        int N = 0;
1349
1350        fprintf_s(out_file, "Exercise 7.4-5\n\n");
```

```
1351      fprintf_s(out_file, "f(x) = x * ln(x) a = 1, b = 2\n");
1352
1353      for (N = 10; N <= 100; N *= 2)
1354      {
1355          double C = ExtCorrTrapezoidalRule(N, a, b, xlnx, dxlnxdx);
1356
1357          if (N == 10)
1358              integ1 = C;
1359          else
1360          {
1361              integ2 = C;
1362              d = fabs(integ1 - integ2);
1363              integ1 = integ2;
1364          }
1365
1366          fprintf_s(out_file, "%s",
1367              "Ext Corrected Trapezoidal Rule = ");
1368          fprintf_s(out_file, "%10.8lf\t", C);
1369          fprintf_s(out_file, "%s", "Delta = ");
1370          fprintf_s(out_file, "%10.8lf\t", d);
1371          fprintf_s(out_file, "%s", "N = ");
1372          fprintf_s(out_file, "%ld\n", N);
1373
1374          if (N > 10 && d < 1.0e-6)
1375              break;
1376      }
1377
1378      d = 0.0;
1379      fprintf_s(out_file, "\n");
1380
1381      for (N = 10; N <= 100; N *= 2)
1382      {
1383          double S = ExtSimpsonsRule(N, a, b, xlnx);
1384
1385          if (N == 10)
1386              integ1 = S;
1387          else
1388          {
1389              integ2 = S;
1390              d = fabs(integ1 - integ2);
1391              integ1 = integ2;
1392          }
1393
1394          fprintf_s(out_file, "%s", "Ext Simpson's Rule = ");
1395          fprintf_s(out_file, "%10.8lf\t", S);
1396          fprintf_s(out_file, "%s", "Delta = ");
1397          fprintf_s(out_file, "%10.8lf\t", d);
1398          fprintf_s(out_file, "%s", "N = ");
1399          fprintf_s(out_file, "%ld\n", N);
1400
1401          if (N > 10 && d < 1.0e-6)
1402              break;
```

```
1403      }
1404
1405      fprintf_s(out_file, "\n");
1406  }
1407
1408  void Exercise7_4_6(FILE* out_file)
1409  {
1410      double a = 1.0, b = 2.0, gi = 0.0;
1411      int N = 0;
1412
1413      fprintf_s(out_file, "Exercise 7.4-6\n\n");
1414      fprintf_s(out_file, "f(x) = x * ln(x) a = 1, b = 2\n");
1415
1416      for (N = 2; N <= 4; N += 2)
1417      {
1418          double* w = allocate_real_vector(0, N);
1419          double* x = allocate_real_vector(0, N);
1420
1421          GenerateGaussianLegendreAbscissasAndWeights(N , x, w);
1422          gi = GLIntegrateAB(N, a, b, x, w, xlnx);
1423          fprintf_s(out_file, "%s", "Gaussian Quadrature  = ");
1424          fprintf_s(out_file, "%+10.8lf\t", gi);
1425          fprintf_s(out_file, "N = %ld\n", N);
1426
1427          free_real_vector(w, 0);
1428          free_real_vector(x, 0);
1429      }
1430
1431      fprintf_s(out_file, "\n");
1432  }
1433
1434  double erfx(double x)
1435  {
1436      double pi = 4.0 * atan(1.0);
1437
1438      return 2.0 * exp(-x * x) / sqrt(pi);
1439  }
1440
1441  void Exercise7_4_7(FILE* out_file)
1442  {
1443      double a = 0.0, b = 0.5, gi = 0.0;
1444      double exact = 0.520499876;
1445      int N = 0;
1446
1447      fprintf_s(out_file, "Exercise 7.4-6\n\n");
1448      fprintf_s(out_file, "f(x) = erf(0.5) a = 1, b = 2\n");
1449
1450      for (N = 2; N <= 4; N += 2)
1451      {
1452          double* w = allocate_real_vector(0, N);
1453          double* x = allocate_real_vector(0, N);
1454
```

```
1455          GenerateGaussianLegendreAbscissasAndWeights(N, x, w);
1456          gi = GLIntegrateAB(N, a, b, x, w, erfx);
1457          fprintf_s(out_file, "%s", "Gaussian Quadrature  = ");
1458          fprintf_s(out_file, "%12.10lf\t", gi);
1459          fprintf_s(out_file, "%s", "Error  = ");
1460          fprintf_s(out_file, "%12.10lf\t", fabs(gi - exact));
1461          fprintf_s(out_file, "N = %ld\n", N);
1462
1463          free_real_vector(w, 0);
1464          free_real_vector(x, 0);
1465      }
1466
1467      fprintf_s(out_file, "\n");
1468  }
1469
1470  double sinx3(double x)
1471  {
1472      return pow(sin(x), 1.0 / 3.0);
1473  }
1474
1475  void Exercise7_4_8(FILE* out_file)
1476  {
1477      double a = 0.0, b = 4.0 * atan(1.0);
1478      int N = 0;
1479
1480      fprintf_s(out_file, "Exercise 7.4-5\n\n");
1481      fprintf_s(out_file, "f(x) = (sin(x)) ^ 1 / 3 a = 0, b = pi\n");
1482
1483      for (N = 5; N <= 20; N += 5)
1484      {
1485          double S = ExtSimpsonsRule(N, a, b, sinx3);
1486
1487          fprintf_s(out_file, "%s",
1488              "Ext Simpson's Rule = ");
1489          fprintf_s(out_file, "%10.8lf\t", S);
1490          fprintf_s(out_file, "%s", "N = ");
1491          fprintf_s(out_file, "%ld\n", N);
1492      }
1493
1494      fprintf_s(out_file, "\n");
1495  }
1496
1497  void Example7_8(FILE* out_file)
1498  {
1499      double AS16 = AdaptiveExtSimpsonsRule(16, 0, 1, sqrt);
1500      double AS32 = AdaptiveExtSimpsonsRule(32, 0, 1, sqrt);
1501      double AS48 = AdaptiveExtSimpsonsRule(48, 0, 1, sqrt);
1502      double AS64 = AdaptiveExtSimpsonsRule(64, 0, 1, sqrt);
1503      double AS80 = AdaptiveExtSimpsonsRule(80, 0, 1, sqrt);
1504      double AS96 = AdaptiveExtSimpsonsRule(96, 0, 1, sqrt);
1505      double ES16 = ExtSimpsonsRule(16, 0, 1, sqrt);
1506      double ES32 = ExtSimpsonsRule(32, 0, 1, sqrt);
```

```
1507      double ES48 = ExtSimpsonsRule(48, 0, 1, sqrt);
1508      double ES64 = ExtSimpsonsRule(64, 0, 1, sqrt);
1509      double ES80 = ExtSimpsonsRule(80, 0, 1, sqrt);
1510      double ES96 = ExtSimpsonsRule(96, 0, 1, sqrt);
1511
1512      fprintf_s(out_file, "Example 7.8\n\n");
1513      fprintf_s(out_file, "f(x) = sqrt(x) a = 0, b = 1\n");
1514      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 16 = ");
1515      fprintf_s(out_file, "%12.10lf\n", AS16);
1516      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 32 = ");
1517      fprintf_s(out_file, "%12.10lf\n", AS32);
1518      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 48 = ");
1519      fprintf_s(out_file, "%12.10lf\n", AS48);
1520      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 64 = ");
1521      fprintf_s(out_file, "%12.10lf\n", AS64);
1522      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 80 = ");
1523      fprintf_s(out_file, "%12.10lf\n", AS80);
1524      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 96 = ");
1525      fprintf_s(out_file, "%12.10lf\n", AS96);
1526
1527      fprintf_s(out_file, "%s", "Extended Simpson's Rule 16     = ");
1528      fprintf_s(out_file, "%12.10lf\n", ES16);
1529      fprintf_s(out_file, "%s", "Extended Simpson's Rule 32     = ");
1530      fprintf_s(out_file, "%12.10lf\n", ES32);
1531      fprintf_s(out_file, "%s", "Extended Simpson's Rule 48     = ");
1532      fprintf_s(out_file, "%12.10lf\n", ES48);
1533      fprintf_s(out_file, "%s", "Extended Simpson's Rule 64     = ");
1534      fprintf_s(out_file, "%12.10lf\n", ES64);
1535      fprintf_s(out_file, "%s", "Extended Simpson's Rule 80     = ");
1536      fprintf_s(out_file, "%12.10lf\n", ES80);
1537      fprintf_s(out_file, "%s", "Extended Simpson's Rule 96     = ");
1538      fprintf_s(out_file, "%12.10lf\n", ES96);
1539      fprintf_s(out_file, "\n");
1540  }
1541
1542  void Exercise7_5_3(FILE* out_file)
1543  {
1544      double AS16 = AdaptiveExtSimpsonsRule(16, 0, 1, Fx);
1545      double AS32 = AdaptiveExtSimpsonsRule(32, 0, 1, Fx);
1546      double AS48 = AdaptiveExtSimpsonsRule(48, 0, 1, Fx);
1547      double AS64 = AdaptiveExtSimpsonsRule(64, 0, 1, Fx);
1548      double AS80 = AdaptiveExtSimpsonsRule(80, 0, 1, Fx);
1549      double ES16 = ExtSimpsonsRule(16, 0, 1, Fx);
1550      double ES32 = ExtSimpsonsRule(32, 0, 1, Fx);
1551      double ES48 = ExtSimpsonsRule(48, 0, 1, Fx);
1552      double ES64 = ExtSimpsonsRule(64, 0, 1, Fx);
1553      double ES80 = ExtSimpsonsRule(80, 0, 1, Fx);
1554
1555      fprintf_s(out_file, "Exercise 7.5-3\n\n");
1556      fprintf_s(out_file, "f(x) = (1 - x * x) ^ 1.5 a = 0, b = 1\n");
1557      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 16 = ");
1558      fprintf_s(out_file, "%12.10lf\n", AS16);
```

```
1559      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 32 = ");
1560      fprintf_s(out_file, "%12.10lf\n", AS32);
1561      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 48 = ");
1562      fprintf_s(out_file, "%12.10lf\n", AS48);
1563      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 64 = ");
1564      fprintf_s(out_file, "%12.10lf\n", AS64);
1565      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 80 = ");
1566      fprintf_s(out_file, "%12.10lf\n", AS80);
1567
1568      fprintf_s(out_file, "%s", "Extended Simpson's Rule 16     = ");
1569      fprintf_s(out_file, "%12.10lf\n", ES16);
1570      fprintf_s(out_file, "%s", "Extended Simpson's Rule 32     = ");
1571      fprintf_s(out_file, "%12.10lf\n", ES32);
1572      fprintf_s(out_file, "%s", "Extended Simpson's Rule 48     = ");
1573      fprintf_s(out_file, "%12.10lf\n", ES48);
1574      fprintf_s(out_file, "%s", "Extended Simpson's Rule 64     = ");
1575      fprintf_s(out_file, "%12.10lf\n", ES64);
1576      fprintf_s(out_file, "%s", "Extended Simpson's Rule 80     = ");
1577      fprintf_s(out_file, "%12.10lf\n", ES80);
1578      fprintf_s(out_file, "\n");
1579  }
1580
1581  double sinxx32(double x)
1582  {
1583      if (x == 0)
1584          return 0.0;
1585      else
1586          return sin(x) / pow(x, 1.5);
1587  }
1588
1589  void Exercise7_5_4(FILE* out_file)
1590  {
1591      double AS16 = AdaptiveExtSimpsonsRule(16, 0, 1, sinxx32);
1592      double AS32 = AdaptiveExtSimpsonsRule(32, 0, 1, sinxx32);
1593      double AS48 = AdaptiveExtSimpsonsRule(48, 0, 1, sinxx32);
1594      double AS64 = AdaptiveExtSimpsonsRule(64, 0, 1, sinxx32);
1595      double AS80 = AdaptiveExtSimpsonsRule(80, 0, 1, sinxx32);
1596      double ES16 = ExtSimpsonsRule(16, 0, 1, sinxx32);
1597      double ES32 = ExtSimpsonsRule(32, 0, 1, sinxx32);
1598      double ES48 = ExtSimpsonsRule(48, 0, 1, sinxx32);
1599      double ES64 = ExtSimpsonsRule(64, 0, 1, sinxx32);
1600      double ES80 = ExtSimpsonsRule(80, 0, 1, sinxx32);
1601
1602      fprintf_s(out_file, "Exercise 7.5-3\n\n");
1603      fprintf_s(out_file, "f(x) = (1 - x * x) ^ 1.5 a = 0, b = 1\n");
1604      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 16 = ");
1605      fprintf_s(out_file, "%12.10lf\n", AS16);
1606      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 32 = ");
1607      fprintf_s(out_file, "%12.10lf\n", AS32);
1608      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 48 = ");
1609      fprintf_s(out_file, "%12.10lf\n", AS48);
1610      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 64 = ");
```

```
1611      fprintf_s(out_file, "%12.10lf\n", AS64);
1612      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 80 = ");
1613      fprintf_s(out_file, "%12.10lf\n", AS80);
1614
1615      fprintf_s(out_file, "%s", "Extended Simpson's Rule 16     = ");
1616      fprintf_s(out_file, "%12.10lf\n", ES16);
1617      fprintf_s(out_file, "%s", "Extended Simpson's Rule 32     = ");
1618      fprintf_s(out_file, "%12.10lf\n", ES32);
1619      fprintf_s(out_file, "%s", "Extended Simpson's Rule 48     = ");
1620      fprintf_s(out_file, "%12.10lf\n", ES48);
1621      fprintf_s(out_file, "%s", "Extended Simpson's Rule 64     = ");
1622      fprintf_s(out_file, "%12.10lf\n", ES64);
1623      fprintf_s(out_file, "%s", "Extended Simpson's Rule 80     = ");
1624      fprintf_s(out_file, "%12.10lf\n", ES80);
1625      fprintf_s(out_file, "\n");
1626  }
1627
1628  void RombergIntegration(
1629      FILE* out_file, double (*f)(double), double a,
1630      double b, int mStart, int nrow)
1631  {
1632      // translated from Conte and de Boor's FORTRAN code
1633
1634      double h = 0.0, ratio = 0.0, sum = 0.0;
1635      double** T = allocate_real_matrix(1, nrow, 1, nrow);
1636      int i = 0, j = 0, k = 0, m = 0;
1637
1638      for (i = 1; i <= nrow; i++)
1639          for (j = 1; j <= nrow; j++)
1640              T[i][j] = 0.0;
1641
1642      m = mStart;
1643      h = (b - a) / m;
1644      sum = 0.5 * (f(a) + f(b));
1645
1646      if (m > 1)
1647      {
1648          for (i = 1; i <= m - 1; i++)
1649              sum += f(a + i * h);
1650      }
1651
1652      T[1][1] = sum * h;
1653
1654      fprintf_s(out_file, "Romberg T-Table\n");
1655      fprintf_s(out_file, "%17.10lf\n", T[1][1]);
1656
1657      if (nrow < 2)
1658          return;
1659
1660      for (k = 2; k <= nrow; k++)
1661      {
1662          h /= 2.0;
```

```
1663            m *= 2;
1664            sum = 0.0;
1665
1666            for (i = 1; i <= m; i += 2)
1667                sum += f(a + i * h);
1668
1669            T[k][1] = T[k - 1][1] / 2.0 + sum * h;
1670
1671            for (j = 1; j <= k - 1; j++)
1672            {
1673                T[k - 1][j] = T[k][j] - T[k - 1][j];
1674                T[k][j + 1] = T[k][j] + T[k - 1][j] / (pow(4.0, j) - 1.0);
1675            }
1676
1677            for (j = 1; j <= k; j++)
1678                fprintf_s(out_file, "%17.10lf\t", T[k][j]);
1679
1680            fprintf_s(out_file, "\n");
1681        }
1682
1683        if (nrow < 3)
1684            return;
1685
1686        fprintf_s(out_file, "Table of ratios\n");
1687
1688        for (k = 1; k <= nrow - 2; k++)
1689        {
1690            for (j = 1; j <= k; j++)
1691            {
1692                if (T[k + 1][j] == 0.0)
1693                    ratio = 0.0;
1694                else
1695                    ratio = T[k][j] / T[k + 1][j];
1696
1697                T[k][j] = ratio;
1698            }
1699
1700            for (j = 1; j <= k; j++)
1701                fprintf_s(out_file, "%6.2lf\t", T[k][j]);
1702
1703            fprintf_s(out_file, "\n");
1704        }
1705
1706        fprintf_s(out_file, "\n");
1707        free_real_matrix(T, 1, nrow, 1);
1708    }
1709
1710    void My_Example(FILE* out_file)
1711    {
1712        double AS16 = AdaptiveExtSimpsonsRule(16, 0, 100, NegSqrExp);
1713        double AS32 = AdaptiveExtSimpsonsRule(32, 0, 100, NegSqrExp);
1714        double AS48 = AdaptiveExtSimpsonsRule(48, 0, 100, NegSqrExp);
```

```
1715      double AS64 = AdaptiveExtSimpsonsRule(64, 0, 100, NegSqrExp);
1716      double AS80 = AdaptiveExtSimpsonsRule(80, 0, 100, NegSqrExp);
1717      double AS96 = AdaptiveExtSimpsonsRule(80, 0, 100, NegSqrExp);
1718      double ES16 = ExtSimpsonsRule(16, 0, 100, NegSqrExp);
1719      double ES32 = ExtSimpsonsRule(32, 0, 100, NegSqrExp);
1720      double ES48 = ExtSimpsonsRule(48, 0, 100, NegSqrExp);
1721      double ES64 = ExtSimpsonsRule(64, 0, 100, NegSqrExp);
1722      double ES80 = ExtSimpsonsRule(80, 0, 100, NegSqrExp);
1723      double ES96 = ExtSimpsonsRule(80, 0, 100, NegSqrExp);
1724      double exact = 0.5 * sqrt(4.0 * atan(1.0));
1725
1726      fprintf_s(out_file, "My Example\n\n");
1727      fprintf_s(out_file, "f(x) = e^(-x*x) a = 0, b = 100\n");
1728      fprintf_s(out_file, "%s", "Exact Value                    = ");
1729      fprintf_s(out_file, "%14.12lf\n", exact);
1730      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 16 = ");
1731      fprintf_s(out_file, "%14.12lf\n", AS16);
1732      fprintf_s(out_file, "%s", "Error 16                       = ");
1733      fprintf_s(out_file, "%14.12lf\n", fabs(AS16 - exact));
1734      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 32 = ");
1735      fprintf_s(out_file, "%14.12lf\n", AS32);
1736      fprintf_s(out_file, "%s", "Error 32                       = ");
1737      fprintf_s(out_file, "%14.12lf\n", fabs(AS32 - exact));
1738      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 48 = ");
1739      fprintf_s(out_file, "%14.12lf\n", AS48);
1740      fprintf_s(out_file, "%s", "Error 48                       = ");
1741      fprintf_s(out_file, "%14.12lf\n", fabs(AS48 - exact));
1742      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 64 = ");
1743      fprintf_s(out_file, "%14.12lf\n", AS64);
1744      fprintf_s(out_file, "%s", "Error 64                       = ");
1745      fprintf_s(out_file, "%14.12lf\n", fabs(AS64 - exact));
1746      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 80 = ");
1747      fprintf_s(out_file, "%14.12lf\n", AS80);
1748      fprintf_s(out_file, "%s", "Error 80                       = ");
1749      fprintf_s(out_file, "%14.12lf\n", fabs(AS80 - exact));
1750      fprintf_s(out_file, "%s", "Adaptive Ext Simpson's Rule 96 = ");
1751      fprintf_s(out_file, "%14.12lf\n", AS96);
1752      fprintf_s(out_file, "%s", "Error 96                       = ");
1753      fprintf_s(out_file, "%14.12lf\n", fabs(AS96 - exact));
1754
1755      fprintf_s(out_file, "%s", "Extended Simpson's Rule 16     = ");
1756      fprintf_s(out_file, "%14.12lf\n", ES16);
1757      fprintf_s(out_file, "%s", "Error 16                       = ");
1758      fprintf_s(out_file, "%14.12lf\n", fabs(ES16 - exact));
1759      fprintf_s(out_file, "%s", "Extended Simpson's Rule 32     = ");
1760      fprintf_s(out_file, "%14.12lf\n", ES32);
1761      fprintf_s(out_file, "%s", "Error 32                       = ");
1762      fprintf_s(out_file, "%14.12lf\n", fabs(ES32 - exact));
1763      fprintf_s(out_file, "%s", "Extended Simpson's Rule 48     = ");
1764      fprintf_s(out_file, "%14.12lf\n", ES48);
1765      fprintf_s(out_file, "%s", "Error 48                       = ");
1766      fprintf_s(out_file, "%14.12lf\n", fabs(ES48 - exact));
```

```
1767      fprintf_s(out_file, "%s", "Extended Simpson's Rule 64     = ");
1768      fprintf_s(out_file, "%14.12lf\n", ES64);
1769      fprintf_s(out_file, "%s", "Error 64                       = ");
1770      fprintf_s(out_file, "%14.12lf\n", fabs(ES64 - exact));
1771      fprintf_s(out_file, "%s", "Extended Simpson's Rule 80     = ");
1772      fprintf_s(out_file, "%14.12lf\n", ES80);
1773      fprintf_s(out_file, "%s", "Error 80                       = ");
1774      fprintf_s(out_file, "%14.12lf\n", fabs(ES80 - exact));
1775      fprintf_s(out_file, "%s", "Extended Simpson's Rule 96     = ");
1776      fprintf_s(out_file, "%14.12lf\n", ES96);
1777      fprintf_s(out_file, "%s", "Error 96                       = ");
1778      fprintf_s(out_file, "%14.12lf\n", fabs(ES96 - exact));
1779      fprintf_s(out_file, "\n");
1780  }
1781
1782  double xsqr(double x)
1783  {
1784      return x * x;
1785  }
1786
1787  double sin101(double x)
1788  {
1789      double pi = 4.0 * atan(1.0);
1790
1791      return sin(101.0 * pi * x);
1792  }
1793
1794  double sin10(double x)
1795  {
1796      double pi = 4.0 * atan(1.0);
1797
1798      return (1.0 + sin(10.0 * pi * x));
1799  }
1800
1801  double abs13(double x)
1802  {
1803      return fabs(x - 1.0 / 3.0);
1804  }
1805
1806  double sinxx(double x)
1807  {
1808      if (x != 0.0)
1809          return sin(x) / x;
1810      else
1811          return 1.0;
1812  }
1813
1814  int main()
1815  {
1816      char cwd[256] = { '\0' };
1817      char out_pathname[256] = { '\0' };
1818      FILE* inp_file = NULL, * out_file = NULL;
```

```
1819      errno_t my_errno = 0;
1820
1821      if (_getcwd(cwd, sizeof(cwd)) == NULL)
1822      {
1823          perror("_getcwd() error");
1824          exit(-1);
1825      }
1826
1827      strcat_s(out_pathname, 256, cwd);
1828      strcat_s(out_pathname, 256, out_filename);
1829
1830      my_errno = fopen_s(&out_file, out_pathname, "w+");
1831
1832      if (my_errno != 0 || out_file == NULL)
1833          exit(errno);
1834
1835      CreateTable7_1(out_file);
1836      fprintf_s(out_file, "\n");
1837      Exercise7_1_1(out_file);
1838      Exercise7_1_2(out_file);
1839      Exercise7_1_3(out_file);
1840
1841      fprintf_s(out_file,
1842          "Exercise 7.1-5\n\n");
1843
1844      Exercise7_1_5(out_file, 16);
1845      Exercise7_1_5(out_file, 24);
1846      Exercise7_1_5(out_file, 32);
1847      Exercise7_1_5(out_file, 48);
1848      Exercise7_1_5(out_file, 64);
1849
1850      Exercise7_2_2(out_file);
1851      Exercise7_2_3(out_file);
1852      Exercise7_2_5(out_file);
1853      Exercise7_2_6(out_file);
1854      Example7_2(out_file);
1855      Example7_2a(out_file);
1856      Example7_3(out_file);
1857      Exercise7_3_4(out_file);
1858      Exercise7_3_9(out_file);
1859      Exercise7_4_3(out_file);
1860      Exercise7_4_4(out_file);
1861      Exercise7_4_5(out_file);
1862      Exercise7_4_6(out_file);
1863      Exercise7_4_7(out_file);
1864      Exercise7_4_8(out_file);
1865      Example7_8(out_file);
1866      Exercise7_5_3(out_file);
1867      Exercise7_5_4(out_file);
1868      fprintf_s(out_file, "Example 7.9\n");
1869      RombergIntegration(
1870          out_file, NegSqrExp, 0.0, 1.0, 2, 6);
```

```
1871      fprintf_s(out_file, "\n");
1872      fprintf_s(out_file, "Exercise 7.7-2 (a)\n");
1873      RombergIntegration(
1874          out_file, xsqr, 0.0, 1.0, 2, 6);
1875      fprintf_s(out_file, "\n");
1876      fprintf_s(out_file, "Exercise 7.7-2 (b)\n");
1877      RombergIntegration(
1878          out_file, sin101, 0.0, 1.0, 1, 6);
1879      fprintf_s(out_file, "\n");
1880      fprintf_s(out_file, "Exercise 7.7-2 (c)\n");
1881      RombergIntegration(
1882          out_file, sin10, 0.0, 1.0, 1, 6);
1883      fprintf_s(out_file, "\n");
1884      fprintf_s(out_file, "Exercise 7.7-2 (d)\n");
1885      RombergIntegration(
1886          out_file, abs13, 0.0, 1.0, 1, 6);
1887      fprintf_s(out_file, "\n");
1888      fprintf_s(out_file, "Exercise 7.7-2 (d)\n");
1889      RombergIntegration(
1890          out_file, abs13, 0.0, 1.0, 3, 6);
1891      fprintf_s(out_file, "\n");
1892      fprintf_s(out_file, "Exercise 7.7-2 (e)\n");
1893      RombergIntegration(
1894          out_file, sqrt, 0.0, 1.0, 2, 6);
1895      fprintf_s(out_file, "\n");
1896      fprintf_s(out_file, "Exercise 7.7-2 (f)\n");
1897      RombergIntegration(
1898          out_file, sinxx, 0.0, 1.0, 2, 6);
1899      fprintf_s(out_file, "\n");
1900      My_Example(out_file);
1901      fclose(out_file);
1902      return 0;
1903  }
```