4-19-2016

# Software Improvements to Parint, a Parallel Integration Software Package

Lawrence Cuneaz
*Western Michigan University*, lawrence.cuneaz@gmail.com

# Software Improvements to the Numerical Integration Software Package ParInt

by

Lawrence Cuneaz
Dillon Daudert
Austin Jones
Kendrick Cline

Western Michigan University
Computer Science Department
May, 2016

# Contents

# Acknowledgements

We acknowledge the support from the National Science Foundation under Award Number 1126438.

# Abstract

The best software is easy to configure and compile, is expandable and is well tested. Developement on the ParInt software package for parallel integration stopped a number of years ago. When handed the software, parts of the package no longer configured or compiled consistently. Furthermore it relied on one random number generator and had no functional testing. The team tuned the auto configuration so that program would configure and build on a current systems, created expandable functionality to add new random number generators and created two functional test packs. Now ParInt can be developed with confidence knowing that it compiles on current systems, multiple random number generators can be used and regressional tests can be run when changes have been made to the source code.

# Preface to ParInt 1.3

ParInt is a software package that solves integration problems numerically. It utilizes multiple processes operating in parallel to solve the problems more quickly. Multivariate vector integrand functions are supported, integrated over hyper-rectangular or simplex regions, using higher-degree quadrature/cubature rules, Quasi-Monte Carlo (QMC), or Monte-Carlo (MC) methods. Various integration parameters control the desired accuracy of the result as well as a limit on the amount of effort taken to solve the problem.

ParInt was developed beginning in 1994 by Elise de Doncker and Ajay Gupta at Western Michigan University, and Alan Genz at Washington State University. The project was joined at later stages by K. Kaugars (primarily with respect to visualization) and J. Kapenga. Furthermore, students Rodger Zanny, Laurentiu Cucos and Shujun Li made considerable contributions. Since its inception, various experimental versions of ParInt, implemented on a variety of platforms, have formed the basis for research in parallel numerical integration, load-balancing algorithms, and distributed data structures (e.g., [4] [5] [13]). The copyright information for ParInt 1.0 is in the copyright section of this document.

The updated version, ParInt 1.3, is the result of a senior design project at Western Michigan University. The group members were Austin Jones, Lawrence Cuneaz, Kendrick Cline and Dillon Daudert. The main areas of development focused on expanding ParInt's usage of random number generators to no longer be reliant on the software package SPRNG (Scalable Parallel Random Number Generators), as well as the inclusion of two different test suites.

The following document covers the changes introduced in ParInt 1.3, documentation on adding new random number generator functions as well as expanding the functionality of the test suites. For more complete information about ParInt, please consult the User's Manual. [13]

# Chapter 1

# Changes in ParInt 1.3

This section begins with details on bug fixes. It also introduces two new features added for ParInt 1.3.

## 1.1 Bug Fixes

One main area that was updated was in the way that ParInt configures itself. The configure script now properly finds and sets the path to the GNU Multiprecision Library when configuring with SPRNG. In addition, many more checks were added to the configure process in an effort to make ParInt more portable to other machines.

An error in the parallel Quasi-Monte Carlo and Monte Carlo methods was fixed. It was found when using ParInt with more recent versions of MPI (i.e. OpenMPI 1.4.3/1.7.3).[11] This turned out to be due to an integer overflow in the relevant code on machines using addresses larger than 32bits.

## 1.2 Generic Random Number Interface

In the original ParInt, the parallel random number generator family library SPRNG (Scalable Parallel Random Number Generators) was required for the Quasi-Monte Carlo and Monte Carlo methods to work in parallel. Over time, the team that developed SPRNG moved on to newer versions that turned out to be incompatible with ParInt. To avoid the issue of being dependent on a single library for its stochastic methods, an interface allowing the swapping in and out of any parallel random number generator desired was created.

Use of the interface is explained in a later chapter. The parallel RNG library Random123 is used as an example. It is also available to users by default, and is enabled by use of a flag passed to the configure script in the same way as is done for SPRNG.

## 1.3 Test Suites

### 1.3.1 Simple Test Suite

This test package is intended to provide integration testing of single and multi-dimensional integrals with all applicable methods provided by ParInt against a set of integrals defined by the user.

It is based on the test package MULTST by Alan Genz [6] (see also [7] [8]). Currently this test pack exercises all of the methods of ParInt except for the ones to calculate integrals over simplices. This testing requires the calculation of the integral by calling ParInt and determining if the results meet the accuracy set by the user.

### 1.3.2 Family Test Suite

This test package is intended to provide comprehensive testing of single and multi-dimensional integrands with all applicable methods provided by ParInt against a set of integrand families. The primary purpose of the family tests is to compare how ParInt's methods compare to each other when evaluating specific types of integrals.

Each method (here also referred to as rule) is tested with the applicable subset of dimensions and with each of the integrand families. By default, 20 samples are performed per test and based on the results, a detailed table for each ParInt-rule/Dimensionality/Integrand-family combination is displayed, detailing results for each combination being tested. The final line of each table is the summarized results for each ParInt-rule/Integrand-family combination across all tested dimensions. These summarized results are stored as a benchmark log file.

Regression testing can then be performed against the benchmark log to determine a pass/fail status for each ParInt-rule/Integrand-family combination in a given environment. The detailed table with results for each ParInt-rule/Dimensionality/Integrand-family combination is displayed, and each ParInt-rule/Integrand-family summary result is compared to the benchmark results to determine and display a pass/fail status for each combination. Once all rules have been tested, the final pass/fail count is displayed.

# Chapter 2

# Customizing and Expanding ParInt

This chapter goes over adding a new parallel random number generator for use in ParInt. It also describes the ways in which a user can add new integrals to both the simple and family test suites.

## 2.1 Adding a Random Number Generator

ParInt 1.3 comes with the option to configure with SPRNG 2.0 and Random123, two established parallel pseudo-random number generator libraries. A third option is also available to the user that allows them to configure ParInt using another random generator. This section will go over the process of adding a new random generator for the parallel stochastic Monte Carlo method. For SPRNG and Random123, the paths to the libraries and headers are set at configure time. If the user adds a new generator, these paths must be set manually at configure time. The second half of this section covers this.

### 2.1.1 The Generic RNG API

There are two functions that ParInt will use when configuring with a new RNG. Both are located in the file `parint_dir/src/rng/set-random.c`. The two functions, `init_g_rng` and `g_rng` have the following definitions:

```
1
2  void init_g_rng(int key)
3  {
4  /* Called first to perform an initialization needed by the
5  random number generator; seed generation */
6      return;
7  }
8
9
10 double g_rng()
11 {
12 /* Returns a uniformly distributed random double in the
13 range [0,1). */
14     return 0.0;
15 }
```

*(Source file is located in the appendices)*

*The* ***init_g_rng*** *function is called first to perform whatever initialization is necessary for* ***g_rng****; for instance, passing a seed to an independent stream. If additional parameters need to be passed to the initialization function, the prototype must also be changed in the* ***parint_dir/src/rng/set-random.h****.*

***g_rng*** *is the function that returns random numbers; it is called by ParInt whenever it needs a new value.*

### 2.1.2 Passing Flags to Configure

*When running the configure script, the user can pass flags to tell ParInt where to find the libraries and headers for new random generators if they are not located in the standard paths.*

*The following flags affect different parts of the compilation process:* ***CPPFLAGS****, flags passed to the C preprocessor (ex:* ***-I*** *dir);* ***LDFLAGS****, flags passed to the linker (ex:* ***-L*** *library);* ***CFLAGS****, flags passed to the compiler itself. The following is an example of the configure step for ParInt if the user wants to compile with a random number generator located in* ***/home/user/rng****, after the functions in* ***set-random.c*** *have been changed as described earlier.*

```
./configure CPPFLAGS='-I/home/user/rng/include' \
    LDFLAGS='-L/home/user/rng/lib/librng.a'     \
    --enable-generic-rng
```

## 2.2 Adding Integrals to the Simple Test Suite

*The following instructions detail the steps that must be taken to add a new integral to the simple test suite for use with ParInt.*

### 2.2.1 Adding an Integrand Function

*Navigate to ParInt1.2D/src/test/ and open the simple_integrands.c file with a text editor and add your own function following the format provided below (see also the user's manual [13]).*

```
int fcn1(int *ndims, pi_base_t x[], int *nfcns,
         pi_base_t funvls[])
{
    if (x[0] != 0 )
    {
        funvls[0] = pi_asin( x[0] ) * pi_log( x[0] );
    }
    else
    {
        funvls[0] = 0.0;
    }
    return 0;
} /* fcn1() */
```

(Source file is located in the appendices)

Decide on a function name not used; fill out the body of the function according to your integrand; copy the above arguments and copy the return type shown above. The x array holds the variables of your function, for example, x[0] = x and x[1] = y. The array element, funvls[0] should be set

to the function evaluation (for a vector function with one component). Make sure to check for any possible undefined function evaluations and set this undefined evaluation to 0. In the example above, the function is not defined at x[0] = 0 so there is an if else statement to catch this. Set the return value to 0 like above.

**NOTE:** If you need to store a value in a variable other than funvls[0], be sure the variable has type pi_base_t so that it is compatible with both double and long double versions of ParInt.

In the same directory open the simple_integrands.h file with a text editor. Add the header of your function to this file and change the NUMOFSTRUCTS defined variable to match the number of integrals you plan to evaluate so that the proper number of structs will be allocated. If you plan on only adding one more integral just increase the NUMOFSTRUCTS defined variable by 1.

### 2.2.2 Creating an integralInfo Struct

Navigate to ParInt1.3/src/test/ and open the simple_structs.c file in a text editor. In this file you will find an integralInfo struct created for each integral labeled /******* Integral ********/. Scroll to the bottom of the file. Place your code before the return statement and after the last end of integral comment, /***** End Integral *****/. First add an allocation section exactly as shown below.

```
/* Allocation */
cases = 10;
rules = calloc(cases, sizeof (int) );
fcn_eval_limit = calloc (cases, sizeof(pi_total_t));
eps_a = calloc (cases, sizeof(pi_base_t));
eps_r = calloc(cases, sizeof(pi_base_t));
eps_a_corr = calloc (cases, sizeof(pi_base_t));
eps_r_corr = calloc(cases, sizeof(pi_base_t));
fcn_eval_limit_corr = calloc (cases, sizeof(pi_total_t));
/* End Allocation */
```

*(Source file is located in the appendices)*

*Here all you need to change is the value of cases if you add or delete a test case for the integral.*

*The next section you need to add is the parallel arrays section to setup the various parallel arrays of test cases (before being placed in the struct). Below is an example of one element in all of the parallel arrays being filled.*

```
/* Parallel Arrays */
rules[3] = 6;
fcn_eval_limit[3] = 4000000;
eps_a[3] = 1.0E-06;
eps_r[3] = 1.0E-06;
eps_a_corr[3] = 100;
eps_r_corr[3] = 100;
fcn_eval_limit_corr[3] = 100;
```

(Source file is located in the appendices)

You can add or delete one of these entire sections as long as array values 0 through (cases-1) are filled, and make sure that you update the cases value to match the number of cases. You can change a rules array element value to values from 1 to 16 only, and the rule has to work with the dimension of the integral you are solving or you will get an error at execution. You can see which methods work for which dimension integrands by looking at Table 1.1 in the ParInt1.2 User Manual [13] but for your convenience all integrals have been set up with a test for each applicable rule. The rest of the parallel array elements can be changed to any value you want as long as the value can be stored as a double or long double depending on how you are executing ParInt. See table 2.1 for information about the values stored in the integralInfo struct.

Lastly add a section to fill your struct as shown below.

```
/* Fill Struct */
structArray[1].name = "fcn2";
structArray[1].fcn = \&fcn2;
structArray[1].dim = 1;
structArray[1].rules = rules;
structArray[1].bounds[0] = 0.0;
structArray[1].bounds[1] = 1.0;
structArray[1].cases = cases;
structArray[1].fcn_eval_limit = fcn_eval_limit;
structArray[1].eps_a = eps_a;
structArray[1].eps_r = eps_r;
structArray[1].eps_a_corr = eps_a_corr;
structArray[1].eps_r_corr = eps_r_corr;
structArray[1].fcn_eval_limit_corr = fcn_eval_limit_corr;
structArray[1].actual = pi_log(2.0) - 2.0;
```

*(Source file is located in the appendices)*

*Do not manipulate the parallel arrays being set in this section. Anything else can be manipulated according to table 2.1 below and the data type seen in the definition of the struct at the top of the file.*

| | |
|---|---|
| name | Function name |
| fcn | Function pointer to the integrand function |
| dim | Dimension of the integral |
| rules | ParInt integration rules, methods (see Table 1.1) |
| bounds[2] | Bounds of the integral: lower bound, upper bound |
| cases | Number of test cases |
| fcn_eval_limit | Function evaluation limit values |
| eps_a | Absolute error tolerances for evaluation |
| eps_r | Relative error tolerances for evaluation |
| eps_a_corr | Values to multiply the eps_a by to allow less strict error |
| eps_r_corr | Values to multiply the eps_r by to allow less strict error |
| fcn_eval_limit_corr | Values to add extra function evaluations to allow for less strict error |
| actual | Actual solution to the integral |

Table 2.1: integralInfo Struct Values

## 2.3 Adding Integrands to the Family Test Suite

*A user may wish to add new integrand families to test with ParInt's integration methods and compare the performance of different methods on these families. There are 6 integrand families provided with the family tests. The following instructions detail the steps that must be taken to add a new integration family to the family tests package for use with ParInt.*

*The integrand needs a very specific form to apply to this test package. It must contain the following properties:*

1. *Both affective and unaffective parameters*
2. *An analytical technique to solve the integral exactly*
3. *The integrand must scale to an arbitrary number of dimensions, including single-dimensional*
4. *A difficulty parameter and exponent must be chosen for the integrand family (the purpose of which is to scale the test results for all integrand families to the same range. See publications on Genz's testpack [6] [7] [8] for more details)*

***Affective parameter:***
*An array of variables Alpha (1 value per dimension), in both the function and integral, which increases integration difficulty as Alpha increases*

***Unaffective parameter:*** *An array of variables Beta (1 value per dimension), in both the function and integral, which should have no effect on integration difficulty, but can be used to produce different integrals ( this is required for sample based testing and statistical summaries of results to be meaningful. )*

*The affective and unaffective parameters are not specified as parameters to the integral or integrand functions, but are instead specified as global variables in the family tests. This is required to get around limitations in the ParInt API. The integrand and integration functions written for an integrand family must use Alpha and Beta internally. For example, see, e.g., genz_function 2-6 in family_integrands.c.*

### 2.3.1 Coding the Integrand

*Add a new integerand function to the end of the file src/test/family_integrands.c. Notice each integrand has an index number in its name; make note of the index associated with this new integrand. This integrand function must conform to all requirements laid out for coding an integrand function in the ParInt1.2 user manual. Add new integrand functions in the same format as in the example below:*

```
int genz_function_1 (int *ndims, pi_base_t x[], int *nfcns,
          pi_base_t funvls[])
{
  const pi_base_t pi = 3.14159265358979323846;
  pi_base_t total;
  funvls[0] = 0.0;
/*
  Oscillatory.
*/
```

```
  total = 2.0 * pi * beta[0] + r8vec_sum ( *ndims, x );
  funvls[0] = cos ( total );
  return 0;
}
```

(Source file is located in the appendices) [1] [7]

Parameter and return value details:

| variable | type | purpose |
|---|---|---|
| ndims | int * | number of dimensions to evaluate over |
| x | pi_base_t[ ] | coordinated of the point to be evaluated |
| nfcns | int * | number of function evaluations to perform |
| funvls | pi_base_t[ ] | double array containing the evaluation results |

### 2.3.2 Modify multst

Modify the function multst() in src/test/family_driver.c and src/test/generateBenchmarks.c as follows:
Within the 3rd nestled loop of multitst.c, alter the break statement shown below to add a new case with the index corresponding to the new integrand, following the format for the other integrands:

```
for ( k = 0; k < nsamp; k++ )
{
    ifail = 1;
    //generate random alpha and beta values for each test sample
    generate_params(ndim, itest, a, b, &total, &dfact, dfclt, exn);
    //Get the exact value of the integral
    value = genz_integral ( itest, ndim, a, b, alpha, beta );

    //call to integration subroutine
    switch(itest){
        case 1:
        (*subrtn) (rule, ndim, a, b, maxpts, &genz_function_1,
        rel_tol, &errest, &finest, &ifail, &eval_count );
        break;
        case 2:
        (*subrtn)(rule, ndim, a, b, maxpts, &genz_function_2,
        rel_tol, &errest, &finest, &ifail, &eval_count );
        break;
        case 3:
        (*subrtn)(rule, ndim, a, b, maxpts, &genz_function_3,
        rel_tol, &errest, &finest, &ifail, &eval_count );
        break;
        case 4:
        (*subrtn)(rule, ndim, a, b, maxpts, &genz_function_4,
        rel_tol, &errest, &finest, &ifail, &eval_count );
        break;
        case 5:
        (*subrtn)(rule, ndim, a, b, maxpts, &genz_function_5,
        rel_tol, &errest, &finest, &ifail, &eval_count );
        break;
        case 6:
        (*subrtn)(rule, ndim, a, b, maxpts, &genz_function_6,
        rel_tol, &errest, &finest, &ifail, &eval_count );
    }
```

*(Source file is located in the appendices) [1] [7]*

### 2.3.3 Modify genz_names

*In src/test/util.c, modify the function genz_names() to add a new case to the switch statement corresponding to the name of the new integrand. Note: the name is limited to 14 total characters. The error message at the end of the function should also be modified to indicate an increment in the valid index range.*

```
char *genz_name ( int indx )
{
  char *name;

  name = ( char * ) malloc ( 14 * sizeof ( char ) );

  if ( indx == 1 )
  {
    strcpy ( name, "Oscillatory   " );
  }
  else if ( indx == 2 )
  {
    strcpy ( name, "Product_Peak " );
  }
  else if ( indx == 3 )
  {
    strcpy ( name, "Corner_Peak  " );
  }
  else if ( indx == 4 )
  {
    strcpy ( name, "Gaussian     " );
  }
  else if ( indx == 5 )
  {
    strcpy ( name, "C0_Function  " );
  }
  else if ( indx == 6 )
  {
    strcpy ( name, "Discontinuous" );
  }
  else
  {
    printf ( "\n" );
    printf ( "  GENZ_NAME - Fatal error!\n" );
    printf ( "  1 <= INDX <= 6 is required.\n" );
    exit ( 1 );
  }
  return name;
}
```

(Source file is located in the appendices) [1] [7]

### 2.3.4 Coding the Integral

Code for the integrand using the global affective and unaffective parameters must be added next. In src/test/family_integrals.c, modify the function genz_integral() to add an if-else statement corresponding to the index of the new integrand family. Below is the first if statement in the genz_integral function:

```
pi_base_t genz_integral ( int indx, int ndim, pi_base_t a[],
  pi_base_t b[], pi_base_t alpha[], pi_base_t beta[] )
{
  pi_base_t ab;
  int *ic;
  int isum;
  int j;
  const pi_base_t pi = 3.14159265358979323846;
  int rank;
  pi_base_t s;
  pi_base_t sgndm;
  pi_base_t total;
  pi_base_t value;
/*
  Oscillatory.
*/
  if ( indx == 1 )
  {
    value = 0.0;
/*
  Generate all sequences of NDIM 0s and 1s.
*/
    rank = 0;
    ic = ( int * ) malloc ( ndim * sizeof ( int ) );

    for ( ; ; )
    {
      tuple_next ( 0, 1, ndim, &rank, ic );

      if ( rank == 0 )
      {
        break;
      }

      total = 2.0 * pi * beta[0];
      for ( j = 0; j < ndim; j++ )
      {
        if ( ic[j] != 1 )
        {
          total = total + alpha[j];
        }
      }

      isum = i4vec_sum ( ndim, ic );

      s = 1 + 2 * ( ( isum / 2 ) * 2 - isum );

      if ( ( ndim % 2 ) == 0 )
      {
        value = value + s * cos ( total );
      }
      else
      {
        value = value + s * sin ( total );
      }
    }
    free ( ic );

    if ( 1 < ( ndim % 4 ) )
```

```
    {
      value = - value;
    }
  }
```

*(Source file is located in the appendices) [1] [7]*

### 2.3.5 Modifying the Test Parameter Structure

*Once the integrand and integral code is added and multst() has been modified, in src/test/family_params.c the global FamilyParams struct and its instantiation must be modified to reflect the increase in number of integrand families to test. Below is the typedef and static declaration of this struct:*

```
struct FamilyParams {
    int tstlim;
    int tstmax;
    int max_eval_lim;
    int n_samp;
    pi_base_t err_tol;
    int nrules;
    int maxdim;
    char* sbname;
    int rules[12];
    int n_dims[12];
    int rule_dims[12][10];
    pi_base_t rule_tol[12][3];
    pi_base_t difclt[6];
    pi_base_t expnts[6];
    int tstfns[6];
  };

struct FamilyParams FP = {
    .tstlim = 6,
    .tstmax = 6,
    .max_eval_lim = 400000,
    .n_samp = 20,
    .err_tol = 1.0E-06,
    .nrules = 11,
    .maxdim = 6,
    .sbname = "ParInt",
    .rules = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16},
    .n_dims = {1, 1, 5, 5, 1, 1, 1, 1, 1, 1, 5},
    .rule_dims = {
        {2},
        {3},
        {2, 3, 4, 6, 8},
        {2, 3, 4, 6, 8},
        {1},
        {1},
        {1},
        {1},
        {1},
        {1},
        {2, 3, 4, 6, 8}},
    .rule_tol = {
        {0.1, 0.1, 1},
        {0.1, 0.1, 1},
```

```
45          {0.1, 0.1, 1},
46          {0.1, 0.1, 1},
47          {0.1, 0.1, 1},
48          {0.1, 0.1, 1},
49          {0.1, 0.1, 1},
50          {0.1, 0.1, 1},
51          {0.1, 0.1, 1},
52          {0.1, 0.1, 1},
53          {0.1, 0.1, 1}},
54      .difclt = { 110.0, 600.0, 600.0, 100.0, 150.0, 100.0 },
55      .expnts = { 1.5, 2.0, 2.0, 1.0, 2.0, 2.0 },
56      .tstfns = { 1, 2, 3, 4, 5, 6 }
57  };
```

(Source file is located in the appendices)

Modify familyParams and FP as follows:

1. Increase tstlim and tstmax by 1.
2. Increase the array initializers for tstfns[], difclt[], and expnts[] by 1.
3. Add the index of the new integrand family to the end of tstfns[] assignment.
4. Add the difficulty value and exponent associated with the integrand family to the end of the assignment of diclt[] and expnts[], respectively.

With these changes complete, the family tests will need to be re-compiled and the benchmarks re-built. At this point regression testing can be performed on the new integrand family.

## 2.4 Modifying Family Test Difficulty Parameters

Various parameters used for the family tests are declared in a static global struct used when generating benchmarks or running regression testing. In order to modify the test difficulty, number of samples, error tolerance, and other parameters, this structure needs to be modified and the family tests re-compiled. Once modified and re-compiled the benchmarks can be rebuilt and regression testing can be performed with the new difficulty parameters.

Summary of the purpose of each variable in the familyParams struct:

| variable | type | purpose |
|---|---|---|
| tstlim | int | The number of integrand families to test |
| tstmax | int | The number of difficulty levels and exponents to test (should be 1 of each per integrand family) |
| max_eval_limit | int | The maximum number of function evaluations allowed by ParInt. This will affect both the run time and accuracy of final results if increased. |
| n_samp | int | The number of samples to take for each test. 20 is used as a default to produce a 95% confidence interval |
| err_tol | pi_base_t | The error tolerance requested from ParInt. Used by ParInt to determine when to stop, and in evaluating the accuracy of ParInt's results vs. the exact Integral values. Increasing this will increase the difficulty and running time of the tests. |
| nrules | int | The number of rules being tested |
| maxdim | int | The highest number of dimensions being tested for any specific rule |
| subname | char* | The name of the integration library being used |
| rules | int[ ] | The indexes corresponding to each ParInt rule being tested (see section on ParInt API in the user manual for details) |
| n_dims | int[ ] | The number of dimensions tested for the rule with corresponding index in Rules[ ] |
| rule_dims | int[ ][ ] | A nested array where each inner array is the list of dimensions to test for the ParInt rule with corresponding index in Rules[ ] |
| rule_tol | pi_base_t[ ][ ] | An array of length-3 arrays, each of which corresponds to the pass/fail tolerance for the three values compared for each ParInt-rule/ Integrand-family combination when regression testing |
| difclt | pi_base_t[ ] | The difficulty values corresponding to the integrand families |
| expnts | pi_base_t[ ] | The difficulty exponents corresponding to the integrand families |
| tstfns | int[] | The indexes of each integrand family being tested |

# Chapter 3

# Executing Test Suites and Interpreting Results

## 3.1 Executing the Simple Test Suite

Follow the directions in ParInt1.3/README to configure and build ParInt. After this the simple test pack executable, simple_test will be created in ParInt1.3/src/test. Execute it with the command $./simple_test

**Note**: The test package can be run in parallel using an mpirun command such as mpirun -np [number of processes] ./simple_test (This command could run multiple processes on one node)

So far, the test package has been executed with parallel (MPI) processes on one node. Note also that the results from MPI runs (of the same program) may be different for different executions. This is likely to happen with ParInt, especially for the adaptive methods.

### 3.1.1 Interpreting Simple Test Results

The results produced by the simple test suite serve the following two purposes:

- Provide information of integration method (rule) performance for each integral/rule pair
- Provide simple Pass/Marginal/Fail results for each test with a final summary of total tests that pass, are marginal, or fail.

Below is a sample output of one integral case execution.

```
1
2 NAME: fcn12
3 RULE: 6
4 ACTABS: 0.000000000000014     ACTREL: 0.000000000000120
5 ACTVAL: 0.117809724509617     RESULT: 0.117809724509603
6 ESTABS: 9.13E-07 (Relative Error 7.75E-06)
7 STATUS: Fcn count: 1995; Rgn count: 95; Fcn count flag: 0
8         Time: 0.000; Time/1M: 0.16792; Time/Region: 0.00000353
9 TEST: Pass
```

Detailed Result Values:

| | |
|---|---|
| NAME | Function name |
| RULE | ParInt rule number corresponding with the Table 1.1 below |
| ACTABS | Actual absolute Error |
| ACTREL | Actual relative Error |
| ACTVAL | Actual integral value of the integral |
| RESULT | Integral value PARINT calculates |
| ESTABS | Estimated absolute Error |
| Fcn count | Number of function evaluations performed by PARINT |
| Rgn count | Number of region evaluations performed during the execution (this is not printed if using QMC or MC) |
| Fcn count flag | Value is 1 if the function count limit is reached, and is 0 otherwise |
| Time | Total time, in seconds, that it took to solve the problem |
| Time/1M | Total time, divided by the number of function evaluations, multiplied by 1,000,000 |
| TEST | Evaluation of test as Pass, Marginal or Fail |

Table 3.1: Detail Result Values

## 3.2 Executing the Family Test Suite

### 3.2.1 Compilation

Follow the directions in ParInt1.3/README to configure and build ParInt. This will produce two executable associated with the family tests, generateBenchmarks and familyTest. The distribution of the family tests comes with a pre-built benchmark log generated on WMU's thor cluster. A copy of the detailed table results produced from testing on thor is also included to allow for detailed comparison between thor and another test enviornment (Note: You can get different results from different MPI runs like mentioned in Section 3.1)

### 3.2.2 generateBenchmarks

Used to run the family tests without the regression component. Displays the detailed table results for all tests, writing summarized results to the file benchmark_log.txt. Re-running generateBenchmarks will overwrite any existing benchmark log file.

### 3.2.3 familyTest:

Used for regression testing. Displays the detailed table results along with a pass/fail result for each ParInt-rule/Integrand-family combination after each method (rule) is tested. Once all methods have run, a final pass/fail count is printed.

### 3.2.4 Re-building the benchmarks

After adding new integrands, altering the test parameters, or before beginning regression testing in a new environment, the benchmark logs will need to be re-generated for regression testing to be meaningful. To re-generate the benchmarks on a new system, simply run generateBenchmarks, benchMarkLog.txt will now contain the updated benchmarks.

### 3.2.5 Interpreting the Family Test results

### 3.2.6 Overview

The family tests are based on Alan Genz's testpack for testing multidimensional integration routines. This testpack has been expanded to run with each of ParInt's integration methods individually. To test a method, 20 similar integrals are generated with random unaffected parameters for each dimension/Integrand-family combination. The integral approximated by the method is compared to the actual integral for the specified parameters. First a table is printed for each family where each row corresponds to the averaged results for the 20 samples, the last row being a summary of results across all dimensions tested. Once all families are tested, a final table with 1 row per family detailing the results across all dimensions tested for each method is displayed.

### 3.2.7 Detailed test results

When either generateBenchmarks or familyTest is run, detailed tables are printed summarizing the results for each 20 samples as confidence intervals. The meaning of these values is described in the following table:

| column name | interpretation |
|---|---|
| N-Dims | the number of dimensions integrated over |
| Integrand family | the integrand family being tested |
| Correct digits estimated | a confidence interval of the number of digits the method has estimated to be correct |
| Correct digits actual | a confidence interval of the number of actually correct digits in the result |
| Reliability | a value summarizing now reliable the results produced by the method are. This isn't how well the method approximated the integral, but rather how much we can trust the results the method produced |
| Wrong Digits | a confidence interval of the number of digits the method got wrong in its integral approximation |
| Integrand evals | a confidence interval of the number of evaluations the method performed on the integrand to produce the result |
| Quality | a score indicating how accurate and reliable the method was. The higher the quality score the better the method works for a given integral |
| Total fails | the number of samples that failed to meet the specified estimated error tolerance before hitting the maximum evaluation limit and throwing the max_eval flag. |

*Example output for rule (method) 3 when running generateBenchmarks:*

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Test Paramaters for ParInt Rule 3:

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

   Showing test results with 20 samples per test.

   Testing the following dimensions:

    2    3    4    6    8

   Difficulty levels   110   600   600   100   150   100
           Exponents1.50000000  2.00000000  2.00000000  1.00000000  2.00000000  2.00000000

   Requested digits = 6           Maximum values = 400000

   Produces variable results with confidence = 0.95861053466796875000

-------------------------------------------------------------------------------------------------------------

 N-Dims Integrand       Correct digits            Relia-  Wrong     Integrand Evals  Quality    Total
        Family          Estimated   Actual        bility  Digits                                Fails

   2   Oscillatory    6.04   6.04   7.61   7.61  1.000  0.00  0.00       1089     1089   8.669        0
   2   Product_Peak   3.59   3.59   8.16   8.16  1.000  0.00  0.00       4785     4785   4.959        0
   2   Corner_Peak    6.08   6.08   6.33   6.33  1.000  0.00  0.00        891      891   7.644        0
   2   Gaussian       6.34   6.34   7.29   7.29  1.000  0.00  0.00        693      693   9.303        0
   2   C0_Function    6.00   6.00   5.26   6.01  0.200  0.16  0.75       5643     6039   4.529        0
   2   Discontinuous  6.06   6.06   5.24   5.24  0.000  0.82  0.82       5049     5049   4.568        0
-------------------------------------------------------------------------------------------------------------
   2   Medians       6.05   6.05   6.05   6.81  1.000  0.00  0.00      2937     2937   6.301

   3   Oscillatory    6.16   6.16   6.45   6.45  1.000  0.00  0.00       4697     4697   6.304        0
   3   Product_Peak   2.74   2.78   6.84   7.56  1.000  0.00  0.00      18711    19635   3.198        0
   3   Corner_Peak    6.11   6.34   4.83   4.92  0.000  1.31  1.48       3927     4081   4.097        0
   3   Gaussian       6.02   6.06   5.87   5.87  0.100  0.15  0.19       5159     5313   5.514        0
   3   C0_Function    6.00   6.00   6.09   6.16  1.000  0.00  0.00     106799   108647   4.310        0
```

```
  3   Discontinuous  6.01   6.04   5.22   5.23  0.000  0.78  0.82      19789    20097   3.911        0
-------------------------------------------------------------------------------------------------------
  3   Medians       6.01   6.05   6.01   6.02  0.550  0.07  0.09    11935    12474   4.204

  4   Oscillatory    6.02   6.02   5.23   5.49  0.000  0.53  0.79      14229    14229   4.405        0
  4   Product_Peak   3.25   3.27   7.13   7.22  1.000  0.00  0.00     105111   112149   3.010        0
  4   Corner_Peak    6.38   6.38   6.21   6.21  0.250  0.17  0.17      23409    23409   5.173        0
  4   Gaussian       6.01   6.05   5.46   5.52  0.000  0.55  0.58      16371    19431   4.299        0
  4   CO_Function    4.71   4.72   5.08   5.09  1.000  0.00  0.00     400095   400401   2.695       20
  4   Discontinuous  6.01   6.02   5.17   5.19  0.000  0.82  0.84      71757    72675   3.419        0
-------------------------------------------------------------------------------------------------------
  4   Medians       6.01   6.02   6.01   5.51  0.125  0.35  0.38    47583    48042   3.859

  6   Oscillatory    6.74   6.74   5.55   5.55  0.000  1.19  1.19        453      453   7.013        0
  6   Product_Peak   5.40   5.41   6.09   6.14  1.000  0.00  0.00     263193   265005   3.651        0
  6   Corner_Peak    6.18   6.18   0.94   0.94  0.000  5.25  5.25       3171     3171   0.465        0
  6   Gaussian       6.00   6.02   5.09   5.09  0.000  0.91  0.93      42129    43035   3.481        0
  6   CO_Function    3.62   3.72   3.63   3.67  0.500  0.00  0.06     400905   400905   1.684       20
  6   Discontinuous  4.07   4.12   2.99   3.07  0.000  1.00  1.11     400905   400905   1.256       20
-------------------------------------------------------------------------------------------------------
  6   Medians       5.70   5.72   5.70   4.38  0.000   0.95  1.02     152661   154020   2.583

  8   Oscillatory    11.10  11.10  7.49   7.49  0.000  3.61  3.61       1105     1105  10.280        0
  8   Product_Peak   7.76   7.76   3.89   3.89  0.000  3.88  3.88       5525     5525   2.773        0
  8   Corner_Peak    11.15  11.15  0.76   0.76  0.000 10.39 10.39       1105     1105   0.443        0
  8   Gaussian       6.00   6.01   5.22   5.53  0.050  0.47  0.78      85085   188955   3.578        0
  8   CO_Function    3.36   3.37   3.37   3.38  1.000  0.00  0.00     403325   405535   1.476       20
  8   Discontinuous  2.16   2.16   0.82   0.89  0.000  1.27  1.33     403325   409955   0.265       20
-------------------------------------------------------------------------------------------------------
  8   Medians       6.88   6.89   6.88   3.63  0.000  2.44  2.47    45305    97240   2.125

-------------------------------------------------------------------------------------------------------

     ParInt Rule 3 Results:

-------------------------------------------------------------------------------------------------------


       Integrand       Correct digits            Relia-  Wrong     Integrand Evals  Quality
       Family          Estimated   Actual        bility  Digits

       Oscillatory   6.16   6.16   6.45   6.45  0.000  0.53  0.79     1105     1105   7.013
       Product_Peak  3.59   3.59   6.84   7.22  1.000  0.00  0.00    18711    19635   3.198
       Corner_Peak   6.18   6.34   4.83   4.92  0.000  1.31  1.48     3171     3171   4.097
       Gaussian      6.01   6.05   5.46   5.53  0.050  0.47  0.58    16371    19431   4.299
       CO_Function   4.71   4.72   5.08   5.09  1.000  0.00  0.00   400095   400401   2.695
       Discontinuous 6.01   6.02   5.17   5.19  0.000  0.82  0.84    71757    72675   3.419
-------------------------------------------------------------------------------------------------------
       Global medians 6.01   6.04   5.32   5.36  0.025  0.50  0.69    17541    19533   3.758
```

### 3.2.8 Regression test results

After each method summary table a pass/fail result for each ParInt-Rule/ Integrand-family is displayed When the familyTest executable runs.The pass/fail result is determined by comparing the three values stored in the benchmark log to the results produced on the current system.

Those values are as follows:

1. Reliability Score
2. Average function Evals
3. Quality

Each of the above values are compared to the corresponding result in benchMarkLog.txt. If the current result using the rule_tol values for the given rule in the familyParams struct is worse than the benchmark result by the specified rule_tol then that test will fail. A test fails if any of the three test parameters fail to fall within their tolerance for the given rule.

Example pass/fail output printed after rule 3 when running familyTest

```
rule # 3 on family Oscillatory    ... OK
```

```
rule # 3 on family Product_Peak  ... OK

rule # 3 on family Corner_Peak   ... OK

rule # 3 on family Gaussian      ... OK

rule # 3 on family C0_Function   ... OK

rule # 3 on family Discontinuous ... OK
```

# Chapter 4

# Integrals and Integrands

In this section all of the simple test suite integrals and all of the family test suite integands have been documented and sourced.

## 4.1 Simple Test Suite Integrals

The parameter $a = 0.5$ unless stated otherwise

1.

$$\int_0^1 \sin^{-1}(x)\ln(x)\,\mathrm{d}x = 2 - \frac{\pi}{2} - \ln(2) \approx -0.26394350735484 \qquad [9]$$

2.

$$\int_0^1 \cos^{-1}(x)\ln(x)\,\mathrm{d}x = \ln(2) - 2 \approx -1.30685281944 \qquad [9]$$

3.

$$\int_0^1 \tan^{-1}(x)\ln(x)\,\mathrm{d}x = \frac{1}{48}(\pi - 12)\pi + \frac{\ln(2)}{2} \approx -0.233207814 \qquad [9]$$

End point singularity at x=0

4.

$$\int_0^1 \cot^{-1}(x)\ln(x)\,\mathrm{d}x = -\frac{1}{48}\pi(12 + \pi) - \frac{\ln(2)}{2} \approx -1.337588512 \qquad [9]$$

End point singularity at x=0

5.

$$\int_0^1 \left|x - \frac{\pi}{4}\right|^a \,\mathrm{d}x = \frac{(1 - \frac{\pi}{4})^{a+1} + (\frac{\pi}{4})^{a+1}}{a+1} \approx 0.164785445 \qquad a = 2 \qquad [12]$$

6.

$$\int_0^1 x^a \ln\left(\frac{1}{x}\right) \mathrm{d}x \;=\; \frac{1}{(a+1)^2} \approx 0.444444444 \qquad \mathrm{Re}(a) > -1 \qquad [12]$$

End point singularity at x=0

7.

$$\int_0^1 \frac{4^{-a}}{(x-\frac{\pi}{4})^2 + 16^{-a}} \,\mathrm{d}x = \tan^{-1}((4-\pi) \times 4^{a-1}) + \tan^{-1}(\pi \times 4^{a-1}) \qquad [12]$$

$\approx 1.4093106235511988028350809 74$

As $a$ increases, the height of the peak at $x = \frac{\pi}{4}$ increases.

8.

$$\int_0^\pi \cos(2^a \sin(x)) \,\mathrm{d}x = \pi \mathrm{J}_0(\sqrt{4^a}) \approx 1.75657172047 \qquad [12]$$

$\mathrm{J}_0(\mathrm{z})$ is the Bessel function of the first kind

As $a$ increases the integrand oscillates more

9.

$$\int_0^1 \left|x - \frac{1}{3}\right|^a \,\mathrm{d}x = \frac{(\frac{2}{3})^{a+1} + (\frac{1}{3})^{a+1}}{a+1} = \frac{1}{9} \approx 0.1111111 \qquad a = 2 \qquad [12]$$

10.

$$\int_0^1 (x^2 - 2x + 3) \,\mathrm{d}x = \frac{7}{3} \approx 2.333333 \qquad [10]$$

11.

$$\int_0^{2\pi} x \sin(30x) \cos(x) \,\mathrm{d}x = \frac{-60\pi}{899} \approx -0.20967 \qquad [10]$$

12.

$$\int_0^{2\pi} x \cos(50x) \sin(30x) \,\mathrm{d}x = \frac{3\pi}{80} \approx 0.117809724509617 \qquad [10]$$

Oscillatory

13.

$$\int_0^1 \frac{1}{\sqrt{x}} \,\mathrm{d}x = 2 \qquad [10]$$

End point singularity at x=0

14.

$$\int_0^1 25e^{-25x}\,\mathrm{d}x = 1 - \frac{1}{e^{25}} \approx 0.999999999986112 \quad [10]$$

15.

$$\int_0^{10} \frac{50}{\pi(1+2500x^2)}\,\mathrm{d}x = \frac{\tan^{-1}(500)}{\pi} \approx 0.49936 \quad [10]$$

16.

$$\int_0^1 \sqrt{x}\,\mathrm{d}x = \frac{2}{3} \approx 0.666667 \quad [2]$$

Derivative singularity at x = 0

17.

$$\int_0^1 x^{\frac{3}{2}}\,\mathrm{d}x = \frac{2}{5} = 0.4 \quad [2]$$

18.

$$\int_0^1 \frac{1}{1+x}\,\mathrm{d}x = \ln(2) \approx 0.69315 \quad [2]$$

19.

$$\int_0^1 \frac{1}{1+x^4}\,\mathrm{d}x = \frac{\pi + 2\coth^{-1}(\sqrt{2})}{4\sqrt{2}} \approx 0.86697 \quad [2]$$

20.

$$\int_0^1 \frac{1}{1+e^x}\,\mathrm{d}x = 1 + \ln(2) - \ln(1+e) \approx 0.37989 \quad [2]$$

21.

$$\int_0^1 \frac{x}{e^x - 1}\,\mathrm{d}x = -\mathrm{Li}_2(\frac{1}{e}) - 1 + \frac{\pi^2}{6} + \ln(e-1) \approx 0.777505 \quad [2]$$

$\mathrm{Li}_2$ is the Spence's function

22.

$$\int_0^1 \frac{2}{2+\sin(10\pi x)}\,\mathrm{d}x = \frac{2}{\sqrt{3}} \approx 1.1547 \quad [2]$$

23.

$$\int_0^{\frac{\pi}{2}} \int_0^{\frac{\pi}{2}} \frac{\sin(y)\sqrt{1-(a^2\sin^2(x))\sin^2(y)}}{1-a^2\sin^2(y)}\,\mathrm{d}x\mathrm{d}y = \frac{\pi}{2\sqrt{1-a^2}} \quad [9]$$

$\approx 1.8137993642342178505940782576 42$

24.

$$\int_0^{\frac{\pi}{2}} \int_0^{\frac{\pi}{2}} \frac{\sin(a)\sin(y)}{\sqrt{1-\sin^2(a)\sin^2(x)\sin^2(y)}}\,\mathrm{d}x\mathrm{d}y = \frac{\pi a}{2} \approx 0.7853981633 \quad [9]$$

25.

$$\int_0^1 \int_0^1 \frac{1}{1-xy}\,\mathrm{d}x\mathrm{d}y = \pi^2/6 \approx 1.6449340668482264364724151 \quad [10]$$

Corner singularity at (1, 1)

26.

$$\int_{-1}^1 \int_{-1}^1 \frac{1}{\sqrt{2-x-y}}\,\mathrm{d}x\mathrm{d}y = -\frac{16}{3}(\sqrt{2}-2) \approx 3.12419 \quad [10]$$

Corner singularity at (1, 1) through singularity at $y = 2 - x$

27.

$$\int_{-1}^1 \int_{-1}^1 \frac{1}{\sqrt{3-x-2y}}\,\mathrm{d}x\mathrm{d}y = \frac{4}{3}(-4-\sqrt{2}+3\sqrt{6}) \quad [10]$$

$\approx 2.5790075546352523277202179998$

Corner singularity at (1, 1), through singularity at $y = \frac{3-x}{2}$

28.

$$\int_{-1}^1 \int_{-1}^1 \left|-\frac{1}{4}+x^2+y^2\right|\,\mathrm{d}x\mathrm{d}y = \frac{5}{3}+\frac{\pi}{16} \approx 1.8630162075160287 \quad [10]$$

29.

$$\int_0^1 \int_0^1 \int_0^1 \frac{1}{\frac{15}{4}-\cos(\pi x)-\cos(\pi y)-\cos(\pi z)}\,\mathrm{d}x\mathrm{d}y\mathrm{d}z \quad [2]$$

$\approx 0.30780727205946079028109352293 5$

30.

$$\int_{-1}^1 \int_{-1}^1 \int_{-1}^1 \sqrt{-\frac{1}{8}+x^2+y^2+z^2}\,\mathrm{d}x\mathrm{d}y\mathrm{d}z \quad [2]$$

$\approx 7.01851683126717$

31.

$f(x) = \cos(\pi(1+6.5x_1+7.5x_2+8.5x_3+7x_4-7x_5))$

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 f(x)\,\mathrm{d}x_1\,\mathrm{d}x_2\,\mathrm{d}x_3\,\mathrm{d}x_4\,\mathrm{d}x_5 = \frac{64}{162435\pi^5} \quad [2]$$

$\approx 0.000001287511$

## 4.2 Family Test Suite Integrands

All integrand families provided with the family tests are attributed to Alan Genz's fortran test-pack for testing multidimensional integrands. [6] [7] [8] Each is defined for an arbitrary dimension.

$f_1(x) = cos(2\pi u_1 + \sum_{i=1}^{n} a_i x_i)$ Oscillatory [8]

$f_2(x) = \prod_{i=1}^{n} (a_i^{-2} + (x_i - u_i)^2)^{-1}$ Product Peak [8]

$f_3(x) = (1 + \sum_{i=1}^{n} a_i x_i)^{-(n+1)}$ Corner Peak [8]

$f_4(x) = e^{(-\sum_{i=1}^{n} a_i^2 (x_i - u_i)^2)}$ Gaussian [8]

$f_5(x) = e^{(-\sum_{i=1}^{n} a_i |x_i - u_i|)}$ $C^0$Function [8]

$f_6(x) = \begin{cases} 0 & \text{if } x_1 > u_1 \text{ or } x_2 > u_2 \\ e^{(\sum_{i=1}^{n} a_i x_i)} & \text{otherwise} \end{cases}$ Discontinuous [8]

# Chapter 5

# Copyright

**ParInt 1.0.**

Type of Work: Computer File
Registration Number / Date: TXu000953383 / 1999-11-10
Title: ParInt 1.0.
Description: 37 p. + computer disk.
Notes: Printout also deposited.
Copyright Claimant: Elise DeDoncker, 1952-, Alan Genz, 1947-, Ajay Gupta, 1961-, & Rodger Zanny, 1968-
Date of Creation: 1999
Copyright Note: C.O. correspondence.
Names: Doncker, Elise D, 1952-
Genz, Alan, 1947-
Gupta, Ajay, 1961-
Zanny, Rodger, 1968-
DeDoncker, Elise

# Bibliography


[1] Burkardt, J. (2016, February 14). TESTPACK Testing Multidimensional Integration Routines. Retrieved February 18, 2016, from http://people.sc.fsu.edu/∼jburkardt/c_src/testpack/testpack.html

[2] Davis, P. J., & Rabinowitz, P. (1975). Methods of numerical integration. New York: Academic Press.

[3] E. de Doncker, F. Yuasa, J. Kapenga, and O. Olagbemi. Scalable Software for Multivariate Integration on Hybrid Platforms. Journal of Physics: Conf. Series, Vol. 640 (2015).

[4] E. de Doncker, K. Kaugars, L. Cucos, and R. Zanny. Current status of the ParInt package for parallel multivariate integration. In Proc. of Computational Particle Physics Symposium (CPP 2001), pages 110–119, 2001.

[5] E. de Doncker, R. Zanny, K. Kaugars, and L. Cucos, "Performance and irregular behavior of adaptive task partitioning," in Lecture Notes in Computer Science, vol. 2074. Springer-Verlag, 2001, pp. 118–127.

[6] Genz, A. MULTST, http://www.math.wsu.edu/math/faculty/genz/homepagev

[7] Genz, A. (1984). Testing Multidimensional Integration Routines. In B. Ford, J. C. Rault, & F. Thomasset (Eds.), Tools, Methods, and Languages for Scientific and Engineering Computation (pp. 81-94). North-Holland.

[8] Genz, A. (1987). A Package for Testing Multiple Integration Subroutines. In P. Keast & G. Fairweather (Eds.), Numerical Integration: Recent Developments, Software and Applications (pp. 337-340). Reidel.

[9] Gradshteĭn, I. S., Ryzhik, I. M., & Jeffrey, A. (1980). Table of integrals, series, and products. New York: Academic Press.

[10] Kuonen, D. (2003). Numerical Integration in S-PLUS or R : A Survey. Journal of Statistical Software J. Stat. Soft., 8(13).

[11] Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/

[12] Piessens, R., deDoncker-Kapenga E., Überhuber C.W, Kahner D (1983). Quadpack a Subroutine Package for Automatic Integration. Berlin, Heidelberg: Springer Berlin Heidelberg.

[13] R. Zanny, E. de Doncker, K. Kaugars, and L. Cucos. PARINT1.2 User's Manual. http://www.cs.wmich.edu/parint.

# Appendices

# Appendix A

# Generic Random Number Source Files

```
/*  File: set-random.c
    ParInt Version: MPI and sequential
    Description: Pseudo-Random Number Generators for ParInt

    Version 1a: 04/08/16  dau  Added Random123 generator

    Version: 02/10/16  dau  Original

*/

#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#include <pi-math.h>
#include <mc.h>

#include <math-adapt.h>

#include <mc-misc.h>
#include <set-random.h>

#ifdef USE_SPRNG
/* Begin SPRNG section */
#define SIMPLE_SPRNG
#define USE_MPI
#include "sprng.h"
/* End SPRNG section */
#endif

#ifdef USE_R123
/* Begin R123 section */
#define USE_MPI
#include <Random123/threefry.h>

threefry2x64_ctr_t r123_ctr = {{0,0}};
threefry2x64_key_t r123_key = {{0,0}};
int r123_index;

/*    Random123 random number generator
 *    Adjusted to be uniform [0, 1)
 *    Reference:
```

```
 *    John K. Salmon, et al. (2011)
 *    "Parallel Random Numbers: as easy as 1, 2, 3"
 * */
void init_r123(int key)
{
    threefry2x64_key_t tmp = {{key, -key}};
    r123_key = tmp;
    r123_index = 0;
}

double r123()
{
    double res;
    /* The maximum value for a 64bit integer*/
    const uint64_t max_64 = 0xFFFFFFFFFFFFFFFF;
    threefry2x64_ctr_t rand = threefry2x64(r123_ctr, r123_key);
    if(r123_index == 0){
        /*Get first number*/
        res = (double) rand.v[0] / max_64;
        r123_index = 1;
        return res;
    }else{
        /*Get second number, increment counter*/
        res = (double) rand.v[1] / max_64;
        r123_index = 0;
        if(r123_ctr.v[0]+1 == 0){
            /*Carry digit */
            r123_ctr.v[0] = 0;
            r123_ctr.v[1]++;
        }else{
            r123_ctr.v[0]++;
        }
        return res;
    }
}
/* End R123 section */
#endif

#ifdef USE_G_RNG
/* User-defined RNG functions */
void init_g_rng(int key)
{/* Called first to perform an initialization needed by the
random number generator; seed generation */
    return;
}
double g_rng()
{/* Returns a uniformly distributed random double in the
range [0,1). */
    return 0.0;
}
/* End user-defined RNG functions*/
#endif

/* this mechanism sets the random generator */
double (*_my_rand)();   /* global var: random generator */
extern void setRandomGenerator(double (*rnd)())
{
    _my_rand = rnd;
}
```

```
102
103
104
105 /*     Uniform (0,1) random number generator */
106 /*     Reference: */
107 /*     L'Ecuyer, Pierre (1996), */
108 /*     "Combined Multiple Recursive Random Number Generators" */
109 /*     Operations Research 44, pp. 816-822. */
110 /*                  INVMP1 = 1/( M1 + 1 ) */
111 /*     Requires some eight digit primes for seeds */
112 #define DEF_X10 15485857
113 #define DEF_X11 17329489
114 #define DEF_X12 36312197
115 #define DEF_X20 55911127
116 #define DEF_X21 75906931
117 #define DEF_X22 96210113
118
119 static int x10 = 11111111;
120 static int x11 = 22222223;
121 static int x12 = 33333335;
122 static int x20 = 44444447;
123 static int x21 = 55555559;
124 static int x22 = 66666661;
125
126 #ifdef QMC_RULE
127 /* Begin QMC unirand definitions */
128
129 x10 = DEF_X10;
130 x11 = DEF_X11;
131 x12 = DEF_X12;
132 x20 = DEF_X20;
133 x21 = DEF_X21;
134 x22 = DEF_X22;
135 /* End QMC unirand definitions */
136 #endif
137
138
139 /* generate random numbers */
140 double uni_rand()
141 {
142     /* System generated locals */
143     double ret_val;
144
145     /* Local variables */
146     int h__, z__, p12, p13, p21, p23;
147
148 /*     Component 1 */
149
150     h__ = x10 / 11714;
151     p13 = (x10 - h__ * 11714) * 183326 - h__ * 2883;
152     h__ = x11 / 33921;
153     p12 = (x11 - h__ * 33921) * 63308 - h__ * 12979;
154     if (p13 < 0) {
155         p13 += 2147483647;
156     }
157     if (p12 < 0) {
158         p12 += 2147483647;
159     }
160     x10 = x11;
```

```
    x11 = x12;
    x12 = p12 - p13;
    if (x12 < 0) {
        x12 += 2147483647;
    }

/*      Component 2 */

    h__ = x20 / 3976;
    p23 = (x20 - h__ * 3976) * 539608 - h__ * 2071;
    h__ = x22 / 24919;
    p21 = (x22 - h__ * 24919) * 86098 - h__ * 7417;
    if (p23 < 0) {
        p23 += 2145483479;
    }
    if (p21 < 0) {
        p21 += 2145483479;
    }
    x20 = x21;
    x21 = x22;
    x22 = p21 - p23;
    if (x22 < 0) {
        x22 += 2145483479;
    }

/*      Combination */

    z__ = x12 - x22;
    if (z__ <= 0) {
        z__ += 2147483647;
    }
    ret_val = z__ * 4.656612873077392578125e-10;
    return ret_val;
}

TRandStatus getRandomDefaultStatus()
{
  TRandStatus ret;
  ret.x[0] = DEF_X10;
  ret.x[1] = DEF_X11;
  ret.x[2] = DEF_X12;
  ret.x[3] = DEF_X20;
  ret.x[4] = DEF_X21;
  ret.x[5] = DEF_X22;
  return ret;
}

/* get the generator status */
TRandStatus getRandomStatus()
{
  TRandStatus ret;
  ret.x[0] = x10;
  ret.x[1] = x11;
  ret.x[2] = x12;
  ret.x[3] = x20;
  ret.x[4] = x21;
  ret.x[5] = x22;
  return ret;
}
```

```
220
221 /* set the generator status */
222 void setRandomStatus(TRandStatus *stat)
223 {
224   x10 = stat->x[0];
225   x11 = stat->x[1];
226   x12 = stat->x[2];
227   x20 = stat->x[3];
228   x21 = stat->x[4];
229   x22 = stat->x[5];
230 }
```

# Appendix B

# Simple Test Pack Source Files

```
/*  File: simple_driver.c
 *  Author: Lawrence Cuneaz
 *  Date: 4/16/2016
 *  Description:
 *  This program builds the array of integral structs that the user has setup
 *  and executes each one for all the cases input by the user by calling ParInt.
 *  This program then prints out the data returned from the execution along with
 *  a pass, marginal or fail rating. After running all of the cases the program
 *  attempts to find system information to print out to the user.
 */
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <pi_options.h>
#include <parint.h>
#include <pi_math.h>
#include "simple_integrands.h"

#ifdef USING_LONG_DOUBLES
#define PI_F(X, Y)  "%"#X "."#Y "Lf  "
#else
#define PI_F(X, Y)  "%"#X "."#Y "f  "
#endif
typedef struct integralInfoTag
{
    char    *name;  // The function name
    int (*fcn) (int *, pi_base_t[], int *, pi_base_t[]);    //The function pointer to the inegrand function to pass ParInt
    int         dim;                    // The dimension of the integral
    pi_base_t   bounds[2];              // bounds[0] = lower bound, bounds[1] = upper bound
    int         cases;                  // The number of different tests for the integral
    int         *rules;                 // The array of ParInt Subroutine rule choices
    pi_total_t  *fcn_eval_limit;        // array of maximum function evaluation points for each rule
    pi_base_t   *eps_a;                 // The array of absolute error tolerances for each rule
    pi_base_t   *eps_r;                 // The array of relative error tolerance for
```

```
     each rule
    pi_base_t   actual;                 // The actual known solution for the
    integral
    pi_base_t   *eps_a_corr;            // The amount to multiply the eps_a to allow
     less strict errors
    pi_base_t   *eps_r_corr;            // The amount to multiply the eps_r to allow
     less strict errors
    pi_total_t  *fcn_eval_limit_corr;   // The amount to add to the fcn_eval_limit
    to allow less strict error
} integralInfo;

//Information need to print output to the user and determine if the test resulted in
     a pass, okay or fail
typedef struct output
{
    char        *name;
    int         rule;
    pi_base_t   result;                 // The result that ParInt calculates
    pi_base_t   actual;                 // The actual known solution for the
    integral
    pi_base_t   estabs;                 // The absolute error estimate
    pi_status_t status;                 // The status structure that ParInt fills
    pi_base_t   eps_a;                  // The absolute error tolerance
    pi_base_t   eps_a_corr;             // The amount to multiply the eps_a to allow
     less strict errors
    pi_base_t   eps_r;                  // The relative error tolerance
    pi_base_t   eps_r_corr;             // The amount to multiply the eps_r to allow
     less strict errors
    pi_total_t  fcn_eval_limit;         //The maximum function evaluations
    pi_total_t  fcn_eval_limit_corr;    // The amount to add to the fcn_eval_limit
    to allow less strict error
} outputType;

integralInfo    *buildStructs(void);
void            print(outputType *data);

outputType      *integrate(outputType *data, char *name, pi_ifcn_t ifcn_p,
                           int ndims, pi_base_t lowBound, pi_base_t upBound,
                           int rule, pi_base_t absErr, pi_base_t relErr,
                           pi_total_t fcn_eval_count, pi_base_t eps_a_corr,
                           pi_base_t eps_r_corr, pi_total_t fcn_eval_limit_corr,
                           pi_base_t solution);

double          totalTime = 0;          // The value to accumulate the time to run
    all user tests
int             passTotal = 0;
int             failTotal = 0;
int             okayTotal = 0;

// Function: main initalizes MPI and ParInt, creates an array of integral structs to
     be processed,

// processes the structs and prints the output of each along with grand total data
int main(int argc, char *argv[])
{
    // Setup MPI and ParInt
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
84      pi_init(MPI_COMM_WORLD);
85
86      // Build integral structs
87      integralInfo    *structArray = buildStructs();
88      outputType      *data = malloc(sizeof(outputType));
89      int             k;
90      int             i;
91
92      // For each struct
93      for (k = 0; k < NUMOFSTRUCTS; k++)
94      {
95          // For each of the cases input by the user execute ParInt
96          for (i = 0; i < structArray[k].cases; i++)
97          {
98              // Call the function to do calculations and return a struct of output
    information
99              data = integrate(data, structArray[k].name, structArray[k].fcn,
100                                 structArray[k].dim, structArray[k].bounds[0],
101                                 structArray[k].bounds[1], structArray[k].rules[i],
102                                 structArray[k].eps_a[i], structArray[k].eps_r[i],
103                                 structArray[k].fcn_eval_limit[i],
104                                 structArray[k].eps_a_corr[i],
105                                 structArray[k].eps_r_corr[i],
106                                 structArray[k].fcn_eval_limit_corr[i],
107                                 structArray[k].actual);
108
109             if (pi_controller(rank))
110             {
111                 print(data);
112             }
113         }
114     }
115
116     printf("Total Time: %.6lf seconds\n", totalTime);
117     printf("Total Passed Tests: %d   Total Marginal Tests: %d    Total Failed Tests:
    %d\n\n",
118        passTotal, okayTotal, failTotal);
119
120     // Print the system information from the version directory in Linux
121     printf("                    System Information                    \n");
122     printf("--------------------------------------------------------------------\n");
123
124     FILE    *file;
125
126     file = fopen("//proc//version", "r");
127     if (file == NULL)
128     {
129         perror("file");
130         exit(1);
131     }
132
133     // When the version file does not exist
134     else
135     {
136         char    buffer[100];
137         while (fgets(buffer, sizeof(buffer), file) != NULL)
138             fputs(buffer, stdout);
139         fclose(file);
140     }
```

```
141
142     free(data);
143     free(structArray);
144     pi_finalize();
145     MPI_Finalize();
146     return 0;
147 }
148
149 // Sets up the region of integration and calls ParInt to integrate
150 outputType *integrate(outputType *data, char *name, pi_ifcn_t ifcn_p, int ndims,
151                       pi_base_t lowBound, pi_base_t upBound, int rule,
152                       pi_base_t absErr, pi_base_t relErr,
153                       pi_total_t fcn_eval_count, pi_base_t eps_a_corr,
154                       pi_base_t eps_r_corr, pi_total_t fcn_eval_limit_corr,
155                       pi_base_t solution)
156 {
157     data->name = strdup(name);
158     data->actual = solution;
159     data->rule = rule;
160     data->eps_a = absErr;
161     data->eps_r = relErr;
162     data->fcn_eval_limit = fcn_eval_count;
163     data->eps_a_corr = eps_a_corr;
164     data->eps_r_corr = eps_r_corr;
165     data->fcn_eval_limit_corr = fcn_eval_limit_corr;
166
167     pi_hregion_t    *hrgn_p;
168     hrgn_p = pi_allocate_hregion(ndims);
169
170     int i;
171     for (i = 0; i < ndims; i++)
172     {
173         hrgn_p->a[i] = lowBound;
174         hrgn_p->b[i] = upBound;
175     }
176
177     pi_integrate(ifcn_p, 1, rule, fcn_eval_count, absErr, relErr, hrgn_p,
178                  PI_RGN_HRECT, &(data->result), &(data->estabs), &(data->status));
179     pi_free_hregion(hrgn_p);
180
181     return data;
182 }
183
184 //Prints formated output to the user for each case
185 void print(outputType *data)
186 {
187     printf("NAME: %-5s\n", data->name);
188     printf("RULE: %d\n", data->rule);
189     printf("ACTABS: "PI_F(-20, 15), pi_abs(data->result - data->actual));
190     printf("ACTREL: "PI_F(0, 15) "\n",
191            pi_abs(data->result - data->actual) / data->actual);
192     printf("ACTVAL: "PI_F(-20, 15), data->actual);
193
194     pi_print_results(&(data->result), &(data->estabs), 1, &(data->status));
195     totalTime += data->status.total_time;
196
197     int          absoluteErrF = 0;
198     int          relativeErrF = 0;
199     int          maxEvalErrF = 0;
```

```
    pi_base_t   estRelErr = data->estabs / data->result;

    // Determine if the test passes
    // Check that the estimated actual error is within eps_a * eps_a_corr
    if (data->estabs < data->eps_a * data->eps_a_corr)
    {
        absoluteErrF = 1;
    }

    // Check that the estimated relative error is within eps_r * eps_r_corr
    if (estRelErr < data->eps_r * data->eps_r_corr)
    {
        relativeErrF = 1;
    }

    // If the maximum number evaluations was not exceeded + the correction factor
    if (data->status.fcn_eval_count <= data->fcn_eval_limit +
        data->fcn_eval_limit_corr)
    {
        maxEvalErrF = 1;
    }

    // Sum the flag values
    int errSum = absoluteErrF + relativeErrF + maxEvalErrF;

    if (errSum == 3)
    {
        printf("%-10s\n", "TEST: Pass\n");
        passTotal++;
    }
    else if (errSum == 2)
    {
        printf("%-10s\n", "TEST: Marginal\n");
        failTotal++;
    }
    else
    {
        printf("%-10s\n", "TEST: Fail\n");
        okayTotal++;
    }
}
```

```
/*  File: simple_integrands.c
 *  Author: Lawrence Cuneaz
 *  Date: 4/16/2016
 *  Description:
 *  This file contains the integrand functions
 */
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <parint.h>
#include <pi - math.h>
#include "simple_integrands.h"

/********************** ONE DIMENSIONAL INTEGRANDS ******************************
    */
int fcn1(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
```

```
{
    if (x[0] != 0)
    {
        funvls[0] = pi_asin(x[0]) * pi_log(x[0]);
    }
    else
    {
        funvls[0] = 0.0;
    }

    return 0;
}   /* fcn1() */

int fcn2(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    if (x[0] != 0)
    {
        funvls[0] = pi_acos(x[0]) * pi_log(x[0]);
    }
    else
    {
        funvls[0] = 0.0;
    }

    return 0;
}   /* fcn2() */

int fcn3(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    if (x[0] != 0)
    {
        funvls[0] = pi_atan(x[0]) * pi_log(x[0]);
    }
    else
    {
        funvls[0] = 0.0;
    }

    return 0;
}   /* fcn3() */

int fcn4(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    if (x[0] != 0)
    {
        funvls[0] = pi_atan(1.0 / x[0]) * pi_log(x[0]);
    }
    else
    {
        funvls[0] = 0.0;
    }

    return 0;
}   /* fcn4() */

int fcn5(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    funvls[0] = pi_pow(pi_abs(x[0] - PI_PI / 4.0), 2.0);
    return 0;
```

```
}   /* fcn5() */

int fcn6(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    if (x[0] != 0)
    {
        funvls[0] = pi_pow(x[0], 0.5) * pi_log(1.0 / x[0]);
    }
    else
    {
        funvls[0] = 0.0;
    }

    return 0;
}   /* fcn6() */

int fcn7(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    pi_base_t   bound = pi_pow(x[0] - (PI_PI / 4.0), 2.0) + pi_pow(16.0,
                                                                   -1.0 * 0.5);

    if (bound != 0)
    {
        funvls[0] = pi_pow(4.0, -1.0 * 0.5) / bound;
    }
    else
    {
        funvls[0] = 0.0;
    }

    return 0;
}   /* fcn7() */

int fcn8(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    funvls[0] = pi_cos(pi_pow(2.0, 0.5) * pi_sin(x[0]));
    return 0;
}   /* fcn8() */

int fcn9(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    funvls[0] = pi_pow(pi_abs(x[0] - 1.0 / 3.0), 2.0);
    return 0;
}   /* fcn9() */

int fcn10(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    funvls[0] = pi_pow(x[0], 2.0) - 2.0 * x[0] + 3.0;
    return 0;
}   /* fcn10() */

int fcn11(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    funvls[0] = x[0] * pi_sin(30.0 * x[0]) * pi_cos(x[0]);
    return 0;
}   /* fcn11() */

int fcn12(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
```

```
135     funvls[0] = x[0] * pi_cos(50.0 * x[0]) * pi_sin(30.0 * x[0]);
136     return 0;
137 }   /* fcn12() */
138 
139 int fcn13(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
140 {
141     if (x[0] > 0)
142     {
143         funvls[0] = 1.0 / pi_sqrt(x[0]);
144     }
145     else
146     {
147         funvls[0] = 0.0;
148     }
149 
150     return 0;
151 }   /* fcn13() */
152 
153 int fcn14(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
154 {
155     funvls[0] = 25.0 * pi_exp(-25.0 * x[0]);
156     return 0;
157 }   /* fcn14() */
158 
159 int fcn15(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
160 {
161     funvls[0] = 50.0 / (PI_PI * (1.0 + 2500.0 * pi_pow(x[0], 2.0)));
162     return 0;
163 }   /* fcn15() */
164 
165 int fcn16(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
166 {
167     if (x[0] >= 0)
168     {
169         funvls[0] = pi_sqrt(x[0]);
170     }
171     else
172     {
173         funvls[0] = 0.0;
174     }
175 
176     return 0;
177 }   /* fcn16() */
178 
179 int fcn17(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
180 {
181     funvls[0] = pi_pow(x[0], 3.0 / 2.0);
182     return 0;
183 }   /* fcn17() */
184 
185 int fcn18(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
186 {
187     funvls[0] = 1.0 / (1.0 + x[0]);
188 
189     return 0;
190 }   /* fcn18() */
191 
192 int fcn19(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
193 {
```

```
194     funvls[0] = 1.0 / (1.0 + pi_pow(x[0], 4.0));
195
196     return 0;
197 }   /* fcn19() */
198
199 int fcn20(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
200 {
201     funvls[0] = 1.0 / (1.0 + pi_pow(M_E, x[0]));
202     return 0;
203 }   /* fcn20() */
204
205 int fcn21(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
206 {
207     if (x[0] != 0)
208     {
209         funvls[0] = x[0] / (pi_pow(M_E, x[0]) - 1.0);
210     }
211     else
212     {
213         funvls[0] = 0;
214     }
215
216     return 0;
217 }   /* fcn21() */
218
219 int fcn22(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
220 {
221     funvls[0] = 2.0 / (2.0 + pi_sin(10.0 * PI_PI * x[0]));
222     return 0;
223 }   /* fcn20() */
224
225 /********************** TWO DIMENSIONAL INTEGRANDS ******************************
    */
226 int fcn23(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
227 {
228     pi_base_t   z = 1.0 - pi_pow(0.5, 2.0) * pi_pow(pi_sin(x[1]), 2.0);
229
230     if (z != 0)
231     {
232         funvls[0] =
233             (
234                 pi_sin(x[1]) *
235                 pi_sqrt(1.0 - (pi_pow(0.5, 2.0) * pi_pow(pi_sin(x[0]), 2.0)) *
236                         pi_pow(pi_sin(x[1]), 2.0))
237             ) /
238             z;
239     }
240     else
241     {
242         funvls[0] = 0;
243     }
244
245     return 0;
246 }   /* fcn23() */
247
248 int fcn24(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
249 {
250     pi_base_t   z =
251         (
```

```
252             1 -
253             pi_pow(pi_sin(0.5), 2.0) *
254             pi_pow(pi_sin(x[0]), 2.0) *
255             pi_pow(pi_sin(x[1]), 2.0)
256         );
257
258     if (z > 0)
259     {
260         funvls[0] = (pi_sin(0.5) * pi_sin(x[1])) / pi_sqrt(z);
261     }
262     else
263     {
264         funvls[0] = 0;
265     }
266
267     return 0;
268 }   /* fcn24() */
269
270 int fcn25(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
271 {TV
272     pi_base_t   z = 1.0 - x[0] * x[1];
273
274     if (z != 0)
275     {
276         funvls[0] = 1.0 / z;
277     }
278     else
279     {
280         funvls[0] = 0.0;
281     }
282
283     return 0;
284 }   /* fcn25() */
285
286 int fcn26(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
287 {
288     pi_base_t   z = pi_sqrt(2.0 - x[0] - x[1]);
289
290     if (z != 0)
291     {
292         funvls[0] = 1.0 / z;
293     }
294     else
295     {
296         funvls[0] = 0.0;
297     }
298
299     return 0;
300 }   /* fcn26() */
301
302 int fcn27(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
303 {
304     pi_base_t   z = pi_sqrt(3.0 - x[0] - 2.0 * x[1]);
305
306     if (z != 0)
307     {
308         funvls[0] = 1 / z;
309     }
310     else
```

```
    {
        funvls[0] = 0.0;
    }

    return 0;
}   /* fcn27() */

int fcn28(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    funvls[0] = pi_abs(-(1.0 / 4.0) + pi_pow(x[0], 2.0) + pi_pow(x[1], 2.0));
    return 0;
}   /* fcn28() */

int fcn29(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    pi_base_t   z = 15.0 /
        4.0 -
        pi_cos(PI_PI * x[0]) -
        pi_cos(PI_PI * x[1]) -
        pi_cos(PI_PI * x[2]);

    if (z != 0)
    {
        funvls[0] = 1.0 / z;
    }
    else
    {
        funvls[0] = 0.0;
    }

    return 0;
}   /* fcn29() */

int fcn30(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    funvls[0] = pi_abs(-(1.0 / 8.0) + pi_pow(x[0], 2) + pi_pow(x[1], 2) + pi_pow(x
    [2], 2));
    return 0;
}   /* fcn30() */

/******************** FIVE DIMENSIONAL INTEGRANDS ******************************/
int fcn31(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
{
    funvls[0] = pi_cos(PI_PI * (1.0 + 6.5 * x[0] + 7.5 * x[1] + 8.5 * x[2] + 7 *
                        x[3] - 7 * x[4]));
    return 0;
}   /* fcn31() */
```

```
/*  File: simple_structs.c
  Author: Lawrence Cuneaz
  Date: 4/16/2016
  Description:
  This file acts as the setup file where the user has to input the different tests
  for each integral and the user has the option of creating multiple tests for each
  integral by extending all of the parallel arrays
*/
#include <stdlib.h>
#include <math.h>
#include <parint.h>
```

```c
#include <pi - math.h>
#include "simple_integrands.h"

typedef struct integralInfoTag
{
    char    *name;  // The function name
    int (*fcn) (int *, pi_base_t[], int *, pi_base_t[]);    //The function pointer to the inegrand function to pass ParInt
    int         dim;                    // The dimension of the integral
    pi_base_t   bounds[2];              // bounds[0] = lower bound, bounds[1] = upper bound
    int         cases;                  // The number of different tests for the integral
    int         *rules;                 // The array of ParInt Subroutine rule choices
    pi_total_t  *fcn_eval_limit;        // array of maximum function evaluation points for each rule
    pi_base_t   *eps_a;                 // The array of absolute error tolerances for each rule
    pi_base_t   *eps_r;                 // The array of relative error tolerance for each rule
    pi_base_t   actual;                 // The actual known solution for the integral
    pi_base_t   *eps_a_corr;            // The amount to multiply the eps_a to allow less strict errors
    pi_base_t   *eps_r_corr;            // The amount to multiply the eps_r to allow less strict errors
    pi_total_t  *fcn_eval_limit_corr;   // The amount to add to the fcn_eval_limit to allow less strict error
} integralInfo;

int (*fcn) (int *, double[], int *, double[]);
integralInfo    *buildStructs();

integralInfo *buildStructs()
{
    integralInfo    *structArray = calloc(NUMOFSTRUCTS, sizeof(integralInfo));

    int             cases;
    int             *rules;
    pi_total_t      *fcn_eval_limit;
    pi_base_t       *eps_a;
    pi_base_t       *eps_r;
    pi_base_t       *eps_a_corr;
    pi_base_t       *eps_r_corr;
    pi_total_t      *fcn_eval_limit_corr;

    /********************** Integral **********************/

    /* Allocation */
    cases = 6;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
    eps_r = calloc(cases, sizeof(pi_base_t));
    eps_a_corr = calloc(cases, sizeof(pi_base_t));
    eps_r_corr = calloc(cases, sizeof(pi_base_t));
    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));

```

```
60      /* End Allocation */
61
62      /* Parallel Arrays */
63      rules[0] = 5;
64      fcn_eval_limit[0] = 4000000;
65      eps_a[0] = 1.0E-06;
66      eps_r[0] = 1.0E-06;
67      eps_a_corr[0] = 100;
68      eps_r_corr[0] = 100;
69      fcn_eval_limit_corr[0] = 100;
70
71      rules[1] = 6;
72      fcn_eval_limit[1] = 4000000;
73      eps_a[1] = 1.0E-06;
74      eps_r[1] = 1.0E-06;
75      eps_a_corr[1] = 100;
76      eps_r_corr[1] = 100;
77      fcn_eval_limit_corr[1] = 100;
78
79      rules[2] = 7;
80      fcn_eval_limit[2] = 4000000;
81      eps_a[2] = 1.0E-06;
82      eps_r[2] = 1.0E-06;
83      eps_a_corr[2] = 100;
84      eps_r_corr[2] = 100;
85      fcn_eval_limit_corr[2] = 100;
86
87      rules[3] = 8;
88      fcn_eval_limit[3] = 4000000;
89      eps_a[3] = 1.0E-06;
90      eps_r[3] = 1.0E-06;
91      eps_a_corr[3] = 100;
92      eps_r_corr[3] = 100;
93      fcn_eval_limit_corr[3] = 100;
94
95      rules[4] = 9;
96      fcn_eval_limit[4] = 4000000;
97      eps_a[4] = 1.0E-06;
98      eps_r[4] = 1.0E-06;
99      eps_a_corr[4] = 100;
100     eps_r_corr[4] = 100;
101     fcn_eval_limit_corr[4] = 100;
102
103     rules[5] = 10;
104     fcn_eval_limit[5] = 4000000;
105     eps_a[5] = 1.0E-06;
106     eps_r[5] = 1.0E-06;
107     eps_a_corr[5] = 100;
108     eps_r_corr[5] = 100;
109     fcn_eval_limit_corr[5] = 100;
110
111     /* End Parallel Arrays */
112
113     /* Fill Struct */
114     structArray[0].name = "fcn1";
115     structArray[0].fcn = &fcn1;
116     structArray[0].dim = 1;
117     structArray[0].rules = rules;
118     structArray[0].bounds[0] = 0.0;
```

```
119     structArray[0].bounds[1] = 1.0;
120     structArray[0].cases = cases;
121     structArray[0].fcn_eval_limit = fcn_eval_limit;
122     structArray[0].eps_a = eps_a;
123     structArray[0].eps_r = eps_r;
124     structArray[0].eps_a_corr = eps_a_corr;
125     structArray[0].eps_r_corr = eps_r_corr;
126     structArray[0].fcn_eval_limit_corr = fcn_eval_limit_corr;
127     structArray[0].actual = 2.0 - PI_PI / 2.0 - pi_log(2.0);
128 
129     /* End Fill Struct */
130 
131     /********************* End Integral ********************/
132 
133     /********************** Integral **********************/
134 
135     /* Allocation */
136     cases = 6;
137     rules = calloc(cases, sizeof(int));
138     fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
139     eps_a = calloc(cases, sizeof(pi_base_t));
140     eps_r = calloc(cases, sizeof(pi_base_t));
141     eps_a_corr = calloc(cases, sizeof(pi_base_t));
142     eps_r_corr = calloc(cases, sizeof(pi_base_t));
143     fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
144 
145     /* End Allocation */
146 
147     /* Parallel Arrays */
148     rules[0] = 5;
149     fcn_eval_limit[0] = 4000000;
150     eps_a[0] = 1.0E-06;
151     eps_r[0] = 1.0E-06;
152     eps_a_corr[0] = 100;
153     eps_r_corr[0] = 100;
154     fcn_eval_limit_corr[0] = 100;
155 
156     rules[1] = 6;
157     fcn_eval_limit[1] = 4000000;
158     eps_a[1] = 1.0E-06;
159     eps_r[1] = 1.0E-06;
160     eps_a_corr[1] = 100;
161     eps_r_corr[1] = 100;
162     fcn_eval_limit_corr[1] = 100;
163 
164     rules[2] = 7;
165     fcn_eval_limit[2] = 4000000;
166     eps_a[2] = 1.0E-06;
167     eps_r[2] = 1.0E-06;
168     eps_a_corr[2] = 100;
169     eps_r_corr[2] = 100;
170     fcn_eval_limit_corr[2] = 100;
171 
172     rules[3] = 8;
173     fcn_eval_limit[3] = 4000000;
174     eps_a[3] = 1.0E-06;
175     eps_r[3] = 1.0E-06;
176     eps_a_corr[3] = 100;
177     eps_r_corr[3] = 100;
```

```
178     fcn_eval_limit_corr[3] = 100;
179
180     rules[4] = 9;
181     fcn_eval_limit[4] = 4000000;
182     eps_a[4] = 1.0E-06;
183     eps_r[4] = 1.0E-06;
184     eps_a_corr[4] = 100;
185     eps_r_corr[4] = 100;
186     fcn_eval_limit_corr[4] = 100;
187
188     rules[5] = 10;
189     fcn_eval_limit[5] = 4000000;
190     eps_a[5] = 1.0E-06;
191     eps_r[5] = 1.0E-06;
192     eps_a_corr[5] = 100;
193     eps_r_corr[5] = 100;
194     fcn_eval_limit_corr[5] = 100;
195
196     /* End Parallel Arrays */
197
198     /* Fill Struct */
199     structArray[1].name = "fcn2";
200     structArray[1].fcn = &fcn2;
201     structArray[1].dim = 1;
202     structArray[1].rules = rules;
203     structArray[1].bounds[0] = 0.0;
204     structArray[1].bounds[1] = 1.0;
205     structArray[1].cases = cases;
206     structArray[1].fcn_eval_limit = fcn_eval_limit;
207     structArray[1].eps_a = eps_a;
208     structArray[1].eps_r = eps_r;
209     structArray[1].eps_a_corr = eps_a_corr;
210     structArray[1].eps_r_corr = eps_r_corr;
211     structArray[1].fcn_eval_limit_corr = fcn_eval_limit_corr;
212     structArray[1].actual = pi_log(2.0) - 2.0;
213
214     /* End Fill Struct */
215
216     /********************** End Integral **********************/
217
218     /********************** Integral **********************/
219
220     /* Allocation */
221     cases = 6;
222     rules = calloc(cases, sizeof(int));
223     fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
224     eps_a = calloc(cases, sizeof(pi_base_t));
225     eps_r = calloc(cases, sizeof(pi_base_t));
226     eps_a_corr = calloc(cases, sizeof(pi_base_t));
227     eps_r_corr = calloc(cases, sizeof(pi_base_t));
228     fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
229
230     /* End Allocation */
231
232     /* Parallel Arrays */
233     rules[0] = 5;
234     fcn_eval_limit[0] = 4000000;
235     eps_a[0] = 1.0E-06;
236     eps_r[0] = 1.0E-06;
```

```
237     eps_a_corr[0] = 100;
238     eps_r_corr[0] = 100;
239     fcn_eval_limit_corr[0] = 100;
240 
241     rules[1] = 6;
242     fcn_eval_limit[1] = 4000000;
243     eps_a[1] = 1.0E-06;
244     eps_r[1] = 1.0E-06;
245     eps_a_corr[1] = 100;
246     eps_r_corr[1] = 100;
247     fcn_eval_limit_corr[1] = 100;
248 
249     rules[2] = 7;
250     fcn_eval_limit[2] = 4000000;
251     eps_a[2] = 1.0E-06;
252     eps_r[2] = 1.0E-06;
253     eps_a_corr[2] = 100;
254     eps_r_corr[2] = 100;
255     fcn_eval_limit_corr[2] = 100;
256 
257     rules[3] = 8;
258     fcn_eval_limit[3] = 4000000;
259     eps_a[3] = 1.0E-06;
260     eps_r[3] = 1.0E-06;
261     eps_a_corr[3] = 100;
262     eps_r_corr[3] = 100;
263     fcn_eval_limit_corr[3] = 100;
264 
265     rules[4] = 9;
266     fcn_eval_limit[4] = 4000000;
267     eps_a[4] = 1.0E-06;
268     eps_r[4] = 1.0E-06;
269     eps_a_corr[4] = 100;
270     eps_r_corr[4] = 100;
271     fcn_eval_limit_corr[4] = 100;
272 
273     rules[5] = 10;
274     fcn_eval_limit[5] = 4000000;
275     eps_a[5] = 1.0E-06;
276     eps_r[5] = 1.0E-06;
277     eps_a_corr[5] = 100;
278     eps_r_corr[5] = 100;
279     fcn_eval_limit_corr[5] = 100;
280 
281     /* End Parallel Arrays */
282 
283     /* Fill Struct */
284     structArray[2].name = "fcn3";
285     structArray[2].fcn = &fcn3;
286     structArray[2].dim = 1;
287     structArray[2].rules = rules;
288     structArray[2].bounds[0] = 0.0;
289     structArray[2].bounds[1] = 1.0;
290     structArray[2].cases = cases;
291     structArray[2].fcn_eval_limit = fcn_eval_limit;
292     structArray[2].eps_a = eps_a;
293     structArray[2].eps_r = eps_r;
294     structArray[2].eps_a_corr = eps_a_corr;
295     structArray[2].eps_r_corr = eps_r_corr;
```

```
296     structArray[2].fcn_eval_limit_corr = fcn_eval_limit_corr;
297     structArray[2].actual = 1.0 /
298         48.0 *
299         (PI_PI - 12.0) *
300         PI_PI +
301         pi_log(2.0) /
302         2.0;
303
304     /* End Fill Struct */
305
306     /********************** End Integral ********************/
307
308     /********************** Integral ***********************/
309
310     /* Allocation */
311     cases = 6;
312     rules = calloc(cases, sizeof(int));
313     fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
314     eps_a = calloc(cases, sizeof(pi_base_t));
315     eps_r = calloc(cases, sizeof(pi_base_t));
316     eps_a_corr = calloc(cases, sizeof(pi_base_t));
317     eps_r_corr = calloc(cases, sizeof(pi_base_t));
318     fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
319
320     /* End Allocation */
321
322     /* Parallel Arrays */
323     rules[0] = 5;
324     fcn_eval_limit[0] = 4000000;
325     eps_a[0] = 1.0E-06;
326     eps_r[0] = 1.0E-06;
327     eps_a_corr[0] = 100;
328     eps_r_corr[0] = 100;
329     fcn_eval_limit_corr[0] = 100;
330
331     rules[1] = 6;
332     fcn_eval_limit[1] = 4000000;
333     eps_a[1] = 1.0E-06;
334     eps_r[1] = 1.0E-06;
335     eps_a_corr[1] = 100;
336     eps_r_corr[1] = 100;
337     fcn_eval_limit_corr[1] = 100;
338
339     rules[2] = 7;
340     fcn_eval_limit[2] = 4000000;
341     eps_a[2] = 1.0E-06;
342     eps_r[2] = 1.0E-06;
343     eps_a_corr[2] = 100;
344     eps_r_corr[2] = 100;
345     fcn_eval_limit_corr[2] = 100;
346
347     rules[3] = 8;
348     fcn_eval_limit[3] = 4000000;
349     eps_a[3] = 1.0E-06;
350     eps_r[3] = 1.0E-06;
351     eps_a_corr[3] = 100;
352     eps_r_corr[3] = 100;
353     fcn_eval_limit_corr[3] = 100;
354
```

```
    rules[4] = 9;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    rules[5] = 10;
    fcn_eval_limit[5] = 4000000;
    eps_a[5] = 1.0E-06;
    eps_r[5] = 1.0E-06;
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[3].name = "fcn4";
    structArray[3].fcn = &fcn4;
    structArray[3].dim = 1;
    structArray[3].rules = rules;
    structArray[3].bounds[0] = 0.0;
    structArray[3].bounds[1] = 1.0;
    structArray[3].cases = cases;
    structArray[3].fcn_eval_limit = fcn_eval_limit;
    structArray[3].eps_a = eps_a;
    structArray[3].eps_r = eps_r;
    structArray[3].eps_a_corr = eps_a_corr;
    structArray[3].eps_r_corr = eps_r_corr;
    structArray[3].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[3].actual = -1.0 /
        48.0 *
        PI_PI *
        (12.0 + PI_PI) -
        pi_log(2.0) /
        2.0;

    /* End Fill Struct */

    /********************* End Integral ********************/

    /********************* Integral ********************/

    /* Allocation */
    cases = 6;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
    eps_r = calloc(cases, sizeof(pi_base_t));
    eps_a_corr = calloc(cases, sizeof(pi_base_t));
    eps_r_corr = calloc(cases, sizeof(pi_base_t));
    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));

    /* End Allocation */

    /* Parallel Arrays */
    rules[0] = 5;
```

```
414     fcn_eval_limit[0] = 4000000;
415     eps_a[0] = 1.0E-06;
416     eps_r[0] = 1.0E-06;
417     eps_a_corr[0] = 100;
418     eps_r_corr[0] = 100;
419     fcn_eval_limit_corr[0] = 100;
420 
421     rules[1] = 6;
422     fcn_eval_limit[1] = 4000000;
423     eps_a[1] = 1.0E-06;
424     eps_r[1] = 1.0E-06;
425     eps_a_corr[1] = 100;
426     eps_r_corr[1] = 100;
427     fcn_eval_limit_corr[1] = 100;
428 
429     rules[2] = 7;
430     fcn_eval_limit[2] = 4000000;
431     eps_a[2] = 1.0E-06;
432     eps_r[2] = 1.0E-06;
433     eps_a_corr[2] = 100;
434     eps_r_corr[2] = 100;
435     fcn_eval_limit_corr[2] = 100;
436 
437     rules[3] = 8;
438     fcn_eval_limit[3] = 4000000;
439     eps_a[3] = 1.0E-06;
440     eps_r[3] = 1.0E-06;
441     eps_a_corr[3] = 100;
442     eps_r_corr[3] = 100;
443     fcn_eval_limit_corr[3] = 100;
444 
445     rules[4] = 9;
446     fcn_eval_limit[4] = 4000000;
447     eps_a[4] = 1.0E-06;
448     eps_r[4] = 1.0E-06;
449     eps_a_corr[4] = 100;
450     eps_r_corr[4] = 100;
451     fcn_eval_limit_corr[4] = 100;
452 
453     rules[5] = 10;
454     fcn_eval_limit[5] = 4000000;
455     eps_a[5] = 1.0E-06;
456     eps_r[5] = 1.0E-06;
457     eps_a_corr[5] = 100;
458     eps_r_corr[5] = 100;
459     fcn_eval_limit_corr[5] = 100;
460 
461     /* End Parallel Arrays */
462 
463     /* Fill Struct */
464     structArray[4].name = "fcn5";
465     structArray[4].fcn = &fcn5;
466     structArray[4].dim = 1;
467     structArray[4].rules = rules;
468     structArray[4].bounds[0] = 0.0;
469     structArray[4].bounds[1] = 1.0;
470     structArray[4].cases = cases;
471     structArray[4].fcn_eval_limit = fcn_eval_limit;
472     structArray[4].eps_a = eps_a;
```

```
473     structArray[4].eps_r = eps_r;
474     structArray[4].eps_a_corr = eps_a_corr;
475     structArray[4].eps_r_corr = eps_r_corr;
476     structArray[4].fcn_eval_limit_corr = fcn_eval_limit_corr;
477     structArray[4].actual =
478         (
479             pi_pow(1.0 - PI_PI / 4.0, 2.0 + 1.0) +
480             pi_pow(PI_PI / 4.0, 2.0 + 1.0)
481         ) /
482         (2.0 + 1.0);
483 
484     /* End Fill Struct */
485 
486     /********************** End Integral ********************/
487 
488     /********************** Integral ***********************/
489 
490     /* Allocation */
491     cases = 6;
492     rules = calloc(cases, sizeof(int));
493     fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
494     eps_a = calloc(cases, sizeof(pi_base_t));
495     eps_r = calloc(cases, sizeof(pi_base_t));
496     eps_a_corr = calloc(cases, sizeof(pi_base_t));
497     eps_r_corr = calloc(cases, sizeof(pi_base_t));
498     fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
499 
500     /* End Allocation */
501 
502     /* Parallel Arrays */
503     rules[0] = 5;
504     fcn_eval_limit[0] = 4000000;
505     eps_a[0] = 1.0E-06;
506     eps_r[0] = 1.0E-06;
507     eps_a_corr[0] = 100;
508     eps_r_corr[0] = 100;
509     fcn_eval_limit_corr[0] = 100;
510 
511     rules[1] = 6;
512     fcn_eval_limit[1] = 4000000;
513     eps_a[1] = 1.0E-06;
514     eps_r[1] = 1.0E-06;
515     eps_a_corr[1] = 100;
516     eps_r_corr[1] = 100;
517     fcn_eval_limit_corr[1] = 100;
518 
519     rules[2] = 7;
520     fcn_eval_limit[2] = 4000000;
521     eps_a[2] = 1.0E-06;
522     eps_r[2] = 1.0E-06;
523     eps_a_corr[2] = 100;
524     eps_r_corr[2] = 100;
525     fcn_eval_limit_corr[2] = 100;
526 
527     rules[3] = 8;
528     fcn_eval_limit[3] = 4000000;
529     eps_a[3] = 1.0E-06;
530     eps_r[3] = 1.0E-06;
531     eps_a_corr[3] = 100;
```

```
    eps_r_corr[3] = 100;
    fcn_eval_limit_corr[3] = 100;

    rules[4] = 9;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    rules[5] = 10;
    fcn_eval_limit[5] = 4000000;
    eps_a[5] = 1.0E-06;
    eps_r[5] = 1.0E-06;
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[5].name = "fcn6";
    structArray[5].fcn = &fcn6;
    structArray[5].dim = 1;
    structArray[5].rules = rules;
    structArray[5].bounds[0] = 0.0;
    structArray[5].bounds[1] = 1.0;
    structArray[5].cases = cases;
    structArray[5].fcn_eval_limit = fcn_eval_limit;
    structArray[5].eps_a = eps_a;
    structArray[5].eps_r = eps_r;
    structArray[5].eps_a_corr = eps_a_corr;
    structArray[5].eps_r_corr = eps_r_corr;
    structArray[5].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[5].actual = 1.0 / (pi_pow(0.5 + 1.0, 2.0));

    /* End Fill Struct */

    /********************** End Integral **********************/

    /********************** Integral **********************/

    /* Allocation */
    cases = 6;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
    eps_r = calloc(cases, sizeof(pi_base_t));
    eps_a_corr = calloc(cases, sizeof(pi_base_t));
    eps_r_corr = calloc(cases, sizeof(pi_base_t));
    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));

    /* End Allocation */

    /* Parallel Arrays */
    rules[0] = 5;
    fcn_eval_limit[0] = 4000000;
    eps_a[0] = 1.0E-06;
```

```
591     eps_r[0] = 1.0E-06;
592     eps_a_corr[0] = 100;
593     eps_r_corr[0] = 100;
594     fcn_eval_limit_corr[0] = 100;
595 
596     rules[1] = 6;
597     fcn_eval_limit[1] = 4000000;
598     eps_a[1] = 1.0E-06;
599     eps_r[1] = 1.0E-06;
600     eps_a_corr[1] = 100;
601     eps_r_corr[1] = 100;
602     fcn_eval_limit_corr[1] = 100;
603 
604     rules[2] = 7;
605     fcn_eval_limit[2] = 4000000;
606     eps_a[2] = 1.0E-06;
607     eps_r[2] = 1.0E-06;
608     eps_a_corr[2] = 100;
609     eps_r_corr[2] = 100;
610     fcn_eval_limit_corr[2] = 100;
611 
612     rules[3] = 8;
613     fcn_eval_limit[3] = 4000000;
614     eps_a[3] = 1.0E-06;
615     eps_r[3] = 1.0E-06;
616     eps_a_corr[3] = 100;
617     eps_r_corr[3] = 100;
618     fcn_eval_limit_corr[3] = 100;
619 
620     rules[4] = 9;
621     fcn_eval_limit[4] = 4000000;
622     eps_a[4] = 1.0E-06;
623     eps_r[4] = 1.0E-06;
624     eps_a_corr[4] = 100;
625     eps_r_corr[4] = 100;
626     fcn_eval_limit_corr[4] = 100;
627 
628     rules[5] = 10;
629     fcn_eval_limit[5] = 4000000;
630     eps_a[5] = 1.0E-06;
631     eps_r[5] = 1.0E-06;
632     eps_a_corr[5] = 100;
633     eps_r_corr[5] = 100;
634     fcn_eval_limit_corr[5] = 100;
635 
636     /* End Parallel Arrays */
637 
638     /* Fill Struct */
639     structArray[6].name = "fcn7";
640     structArray[6].fcn = &fcn7;
641     structArray[6].dim = 1;
642     structArray[6].rules = rules;
643     structArray[6].bounds[0] = 0.0;
644     structArray[6].bounds[1] = 1.0;
645     structArray[6].cases = cases;
646     structArray[6].fcn_eval_limit = fcn_eval_limit;
647     structArray[6].eps_a = eps_a;
648     structArray[6].eps_r = eps_r;
649     structArray[6].eps_a_corr = eps_a_corr;
```

```
650     structArray[6].eps_r_corr = eps_r_corr;
651     structArray[6].fcn_eval_limit_corr = fcn_eval_limit_corr;
652     structArray[6].actual = pi_atan((4.0 - PI_PI) * pi_pow(4.0, 0.5 - 1.0)) +
        pi_atan(PI_PI * pi_pow(4.0, 0.5 - 1.0));
653
654     /* End Fill Struct */
655
656     /********************* End Integral *********************/
657
658     /********************** Integral ***********************/
659
660     /* Allocation */
661     cases = 6;
662     rules = calloc(cases, sizeof(int));
663     fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
664     eps_a = calloc(cases, sizeof(pi_base_t));
665     eps_r = calloc(cases, sizeof(pi_base_t));
666     eps_a_corr = calloc(cases, sizeof(pi_base_t));
667     eps_r_corr = calloc(cases, sizeof(pi_base_t));
668     fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
669
670     /* End Allocation */
671
672     /* Parallel Arrays */
673     rules[0] = 5;
674     fcn_eval_limit[0] = 4000000;
675     eps_a[0] = 1.0E-06;
676     eps_r[0] = 1.0E-06;
677     eps_a_corr[0] = 100;
678     eps_r_corr[0] = 100;
679     fcn_eval_limit_corr[0] = 100;
680
681     rules[1] = 6;
682     fcn_eval_limit[1] = 4000000;
683     eps_a[1] = 1.0E-06;
684     eps_r[1] = 1.0E-06;
685     eps_a_corr[1] = 100;
686     eps_r_corr[1] = 100;
687     fcn_eval_limit_corr[1] = 100;
688
689     rules[2] = 7;
690     fcn_eval_limit[2] = 4000000;
691     eps_a[2] = 1.0E-06;
692     eps_r[2] = 1.0E-06;
693     eps_a_corr[2] = 100;
694     eps_r_corr[2] = 100;
695     fcn_eval_limit_corr[2] = 100;
696
697     rules[3] = 8;
698     fcn_eval_limit[3] = 4000000;
699     eps_a[3] = 1.0E-06;
700     eps_r[3] = 1.0E-06;
701     eps_a_corr[3] = 100;
702     eps_r_corr[3] = 100;
703     fcn_eval_limit_corr[3] = 100;
704
705     rules[4] = 9;
706     fcn_eval_limit[4] = 4000000;
707     eps_a[4] = 1.0E-06;
```

```
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    rules[5] = 10;
    fcn_eval_limit[5] = 4000000;
    eps_a[5] = 1.0E-06;
    eps_r[5] = 1.0E-06;
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[7].name = "fcn8";
    structArray[7].fcn = &fcn8;
    structArray[7].dim = 1;
    structArray[7].rules = rules;
    structArray[7].bounds[0] = 0.0;
    structArray[7].bounds[1] = PI_PI;
    structArray[7].cases = cases;
    structArray[7].fcn_eval_limit = fcn_eval_limit;
    structArray[7].eps_a = eps_a;
    structArray[7].eps_r = eps_r;
    structArray[7].eps_a_corr = eps_a_corr;
    structArray[7].eps_r_corr = eps_r_corr;
    structArray[7].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[7].actual = PI_PI * pi_j0(pi_sqrt(pi_pow(4.0, 0.5)));

    /* End Fill Struct */

    /********************** End Integral ********************/

    /********************** Integral ***********************/

    /* Allocation */
    cases = 6;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
    eps_r = calloc(cases, sizeof(pi_base_t));
    eps_a_corr = calloc(cases, sizeof(pi_base_t));
    eps_r_corr = calloc(cases, sizeof(pi_base_t));
    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));

    /* End Allocation */

    /* Parallel Arrays */
    rules[0] = 5;
    fcn_eval_limit[0] = 4000000;
    eps_a[0] = 1.0E-06;
    eps_r[0] = 1.0E-06;
    eps_a_corr[0] = 100;
    eps_r_corr[0] = 100;
    fcn_eval_limit_corr[0] = 100;

    rules[1] = 6;
```

```
    fcn_eval_limit[1] = 4000000;
    eps_a[1] = 1.0E-06;
    eps_r[1] = 1.0E-06;
    eps_a_corr[1] = 100;
    eps_r_corr[1] = 100;
    fcn_eval_limit_corr[1] = 100;

    rules[2] = 7;
    fcn_eval_limit[2] = 4000000;
    eps_a[2] = 1.0E-06;
    eps_r[2] = 1.0E-06;
    eps_a_corr[2] = 100;
    eps_r_corr[2] = 100;
    fcn_eval_limit_corr[2] = 100;

    rules[3] = 8;
    fcn_eval_limit[3] = 4000000;
    eps_a[3] = 1.0E-06;
    eps_r[3] = 1.0E-06;
    eps_a_corr[3] = 100;
    eps_r_corr[3] = 100;
    fcn_eval_limit_corr[3] = 100;

    rules[4] = 9;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    rules[5] = 10;
    fcn_eval_limit[5] = 4000000;
    eps_a[5] = 1.0E-06;
    eps_r[5] = 1.0E-06;
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[8].name = "fcn9";
    structArray[8].fcn = &fcn9;
    structArray[8].dim = 1;
    structArray[8].rules = rules;
    structArray[8].bounds[0] = 0.0;
    structArray[8].bounds[1] = 1.0;
    structArray[8].cases = cases;
    structArray[8].fcn_eval_limit = fcn_eval_limit;
    structArray[8].eps_a = eps_a;
    structArray[8].eps_r = eps_r;
    structArray[8].eps_a_corr = eps_a_corr;
    structArray[8].eps_r_corr = eps_r_corr;
    structArray[8].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[8].actual =
        (
            pi_pow(2.0 / 3.0, 2 + 1.0) +
            pi_pow(1.0 / 3.0, 2.0 + 1.0)
```

```
826         ) /
827         (2.0 + 1.0);
828 
829     /* End Fill Struct */
830 
831     /********************* Integral ********************/
832 
833     /****************** End Integral *******************/
834 
835     /* Allocation */
836     cases = 6;
837     rules = calloc(cases, sizeof(int));
838     fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
839     eps_a = calloc(cases, sizeof(pi_base_t));
840     eps_r = calloc(cases, sizeof(pi_base_t));
841     eps_a_corr = calloc(cases, sizeof(pi_base_t));
842     eps_r_corr = calloc(cases, sizeof(pi_base_t));
843     fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
844 
845     /* End Allocation */
846 
847     /* Parallel Arrays */
848     rules[0] = 5;
849     fcn_eval_limit[0] = 4000000;
850     eps_a[0] = 1.0E-06;
851     eps_r[0] = 1.0E-06;
852     eps_a_corr[0] = 100;
853     eps_r_corr[0] = 100;
854     fcn_eval_limit_corr[0] = 100;
855 
856     rules[1] = 6;
857     fcn_eval_limit[1] = 4000000;
858     eps_a[1] = 1.0E-06;
859     eps_r[1] = 1.0E-06;
860     eps_a_corr[1] = 100;
861     eps_r_corr[1] = 100;
862     fcn_eval_limit_corr[1] = 100;
863 
864     rules[2] = 7;
865     fcn_eval_limit[2] = 4000000;
866     eps_a[2] = 1.0E-06;
867     eps_r[2] = 1.0E-06;
868     eps_a_corr[2] = 100;
869     eps_r_corr[2] = 100;
870     fcn_eval_limit_corr[2] = 100;
871 
872     rules[3] = 8;
873     fcn_eval_limit[3] = 4000000;
874     eps_a[3] = 1.0E-06;
875     eps_r[3] = 1.0E-06;
876     eps_a_corr[3] = 100;
877     eps_r_corr[3] = 100;
878     fcn_eval_limit_corr[3] = 100;
879 
880     rules[4] = 9;
881     fcn_eval_limit[4] = 4000000;
882     eps_a[4] = 1.0E-06;
883     eps_r[4] = 1.0E-06;
884     eps_a_corr[4] = 100;
```

```
885     eps_r_corr[4] = 100;
886     fcn_eval_limit_corr[4] = 100;
887 
888     rules[5] = 10;
889     fcn_eval_limit[5] = 4000000;
890     eps_a[5] = 1.0E-06;
891     eps_r[5] = 1.0E-06;
892     eps_a_corr[5] = 100;
893     eps_r_corr[5] = 100;
894     fcn_eval_limit_corr[5] = 100;
895 
896     /* End Parallel Arrays */
897 
898     /* Fill Struct */
899     structArray[9].name = "fcn10";
900     structArray[9].fcn = &fcn10;
901     structArray[9].dim = 1;
902     structArray[9].rules = rules;
903     structArray[9].bounds[0] = 0.0;
904     structArray[9].bounds[1] = 1.0;
905     structArray[9].cases = cases;
906     structArray[9].fcn_eval_limit = fcn_eval_limit;
907     structArray[9].eps_a = eps_a;
908     structArray[9].eps_r = eps_r;
909     structArray[9].eps_a_corr = eps_a_corr;
910     structArray[9].eps_r_corr = eps_r_corr;
911     structArray[9].fcn_eval_limit_corr = fcn_eval_limit_corr;
912     structArray[9].actual = 7.0 / 3.0;
913 
914     /* End Fill Struct */
915 
916     /********************* End Integral ********************/
917 
918     /********************** Integral **********************/
919 
920     /* Allocation */
921     cases = 6;
922     rules = calloc(cases, sizeof(int));
923     fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
924     eps_a = calloc(cases, sizeof(pi_base_t));
925     eps_r = calloc(cases, sizeof(pi_base_t));
926     eps_a_corr = calloc(cases, sizeof(pi_base_t));
927     eps_r_corr = calloc(cases, sizeof(pi_base_t));
928     fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
929 
930     /* End Allocation */
931 
932     /* Parallel Arrays */
933     rules[0] = 5;
934     fcn_eval_limit[0] = 4000000;
935     eps_a[0] = 1.0E-06;
936     eps_r[0] = 1.0E-06;
937     eps_a_corr[0] = 100;
938     eps_r_corr[0] = 100;
939     fcn_eval_limit_corr[0] = 100;
940 
941     rules[1] = 6;
942     fcn_eval_limit[1] = 4000000;
943     eps_a[1] = 1.0E-06;
```

```
944     eps_r[1] = 1.0E-06;
945     eps_a_corr[1] = 100;
946     eps_r_corr[1] = 100;
947     fcn_eval_limit_corr[1] = 100;
948 
949     rules[2] = 7;
950     fcn_eval_limit[2] = 4000000;
951     eps_a[2] = 1.0E-06;
952     eps_r[2] = 1.0E-06;
953     eps_a_corr[2] = 100;
954     eps_r_corr[2] = 100;
955     fcn_eval_limit_corr[2] = 100;
956 
957     rules[3] = 8;
958     fcn_eval_limit[3] = 4000000;
959     eps_a[3] = 1.0E-06;
960     eps_r[3] = 1.0E-06;
961     eps_a_corr[3] = 100;
962     eps_r_corr[3] = 100;
963     fcn_eval_limit_corr[3] = 100;
964 
965     rules[4] = 9;
966     fcn_eval_limit[4] = 4000000;
967     eps_a[4] = 1.0E-06;
968     eps_r[4] = 1.0E-06;
969     eps_a_corr[4] = 100;
970     eps_r_corr[4] = 100;
971     fcn_eval_limit_corr[4] = 100;
972 
973     rules[5] = 10;
974     fcn_eval_limit[5] = 4000000;
975     eps_a[5] = 1.0E-06;
976     eps_r[5] = 1.0E-06;
977     eps_a_corr[5] = 100;
978     eps_r_corr[5] = 100;
979     fcn_eval_limit_corr[5] = 100;
980 
981     /* End Parallel Arrays */
982 
983     /* Fill Struct */
984     structArray[10].name = "fcn11";
985     structArray[10].fcn = &fcn11;
986     structArray[10].dim = 1;
987     structArray[10].rules = rules;
988     structArray[10].bounds[0] = 0.0;
989     structArray[10].bounds[1] = 2.0 * PI_PI;
990     structArray[10].cases = cases;
991     structArray[10].fcn_eval_limit = fcn_eval_limit;
992     structArray[10].eps_a = eps_a;
993     structArray[10].eps_r = eps_r;
994     structArray[10].eps_a_corr = eps_a_corr;
995     structArray[10].eps_r_corr = eps_r_corr;
996     structArray[10].fcn_eval_limit_corr = fcn_eval_limit_corr;
997     structArray[10].actual = -60 * PI_PI / 899.0;
998 
999     /* End Fill Struct */
1000 
1001     /********************* End Integral ********************/
1002 
```

```
1003    /********************** Integral **********************/
1004
1005    /* Allocation */
1006    cases = 6;
1007    rules = calloc(cases, sizeof(int));
1008    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
1009    eps_a = calloc(cases, sizeof(pi_base_t));
1010    eps_r = calloc(cases, sizeof(pi_base_t));
1011    eps_a_corr = calloc(cases, sizeof(pi_base_t));
1012    eps_r_corr = calloc(cases, sizeof(pi_base_t));
1013    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
1014
1015    /* End Allocation */
1016
1017    /* Parallel Arrays */
1018    rules[0] = 5;
1019    fcn_eval_limit[0] = 4000000;
1020    eps_a[0] = 1.0E-06;
1021    eps_r[0] = 1.0E-06;
1022    eps_a_corr[0] = 100;
1023    eps_r_corr[0] = 100;
1024    fcn_eval_limit_corr[0] = 100;
1025
1026    rules[1] = 6;
1027    fcn_eval_limit[1] = 4000000;
1028    eps_a[1] = 1.0E-06;
1029    eps_r[1] = 1.0E-06;
1030    eps_a_corr[1] = 100;
1031    eps_r_corr[1] = 100;
1032    fcn_eval_limit_corr[1] = 100;
1033
1034    rules[2] = 7;
1035    fcn_eval_limit[2] = 4000000;
1036    eps_a[2] = 1.0E-06;
1037    eps_r[2] = 1.0E-06;
1038    eps_a_corr[2] = 100;
1039    eps_r_corr[2] = 100;
1040    fcn_eval_limit_corr[2] = 100;
1041
1042    rules[3] = 8;
1043    fcn_eval_limit[3] = 4000000;
1044    eps_a[3] = 1.0E-06;
1045    eps_r[3] = 1.0E-06;
1046    eps_a_corr[3] = 100;
1047    eps_r_corr[3] = 100;
1048    fcn_eval_limit_corr[3] = 100;
1049
1050    rules[4] = 9;
1051    fcn_eval_limit[4] = 4000000;
1052    eps_a[4] = 1.0E-06;
1053    eps_r[4] = 1.0E-06;
1054    eps_a_corr[4] = 100;
1055    eps_r_corr[4] = 100;
1056    fcn_eval_limit_corr[4] = 100;
1057
1058    rules[5] = 10;
1059    fcn_eval_limit[5] = 4000000;
1060    eps_a[5] = 1.0E-06;
1061    eps_r[5] = 1.0E-06;
```

```
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[11].name = "fcn12";
    structArray[11].fcn = &fcn12;
    structArray[11].dim = 1;
    structArray[11].rules = rules;
    structArray[11].bounds[0] = 0.0;
    structArray[11].bounds[1] = 2.0 * PI_PI;
    structArray[11].cases = cases;
    structArray[11].fcn_eval_limit = fcn_eval_limit;
    structArray[11].eps_a = eps_a;
    structArray[11].eps_r = eps_r;
    structArray[11].eps_a_corr = eps_a_corr;
    structArray[11].eps_r_corr = eps_r_corr;
    structArray[11].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[11].actual = 3 * PI_PI / 80;

    /* End Fill Struct */

    /********************** End Integral ********************/

    /*********************** Integral ***********************/

    /* Allocation */
    cases = 6;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
    eps_r = calloc(cases, sizeof(pi_base_t));
    eps_a_corr = calloc(cases, sizeof(pi_base_t));
    eps_r_corr = calloc(cases, sizeof(pi_base_t));
    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));

    /* End Allocation */

    /* Parallel Arrays */
    rules[0] = 5;
    fcn_eval_limit[0] = 4000000;
    eps_a[0] = 1.0E-06;
    eps_r[0] = 1.0E-06;
    eps_a_corr[0] = 100;
    eps_r_corr[0] = 100;
    fcn_eval_limit_corr[0] = 100;

    rules[1] = 6;
    fcn_eval_limit[1] = 4000000;
    eps_a[1] = 1.0E-06;
    eps_r[1] = 1.0E-06;
    eps_a_corr[1] = 100;
    eps_r_corr[1] = 100;
    fcn_eval_limit_corr[1] = 100;

    rules[2] = 7;
    fcn_eval_limit[2] = 4000000;
```

```
    eps_a[2] = 1.0E-06;
    eps_r[2] = 1.0E-06;
    eps_a_corr[2] = 100;
    eps_r_corr[2] = 100;
    fcn_eval_limit_corr[2] = 100;

    rules[3] = 8;
    fcn_eval_limit[3] = 4000000;
    eps_a[3] = 1.0E-06;
    eps_r[3] = 1.0E-06;
    eps_a_corr[3] = 100;
    eps_r_corr[3] = 100;
    fcn_eval_limit_corr[3] = 100;

    rules[4] = 9;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    rules[5] = 10;
    fcn_eval_limit[5] = 4000000;
    eps_a[5] = 1.0E-06;
    eps_r[5] = 1.0E-06;
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[12].name = "fcn13";
    structArray[12].fcn = &fcn13;
    structArray[12].dim = 1;
    structArray[12].rules = rules;
    structArray[12].bounds[0] = 0.0;
    structArray[12].bounds[1] = 1.0;
    structArray[12].cases = cases;
    structArray[12].fcn_eval_limit = fcn_eval_limit;
    structArray[12].eps_a = eps_a;
    structArray[12].eps_r = eps_r;
    structArray[12].eps_a_corr = eps_a_corr;
    structArray[12].eps_r_corr = eps_r_corr;
    structArray[12].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[12].actual = 2.0;

    /* End Fill Struct */

    /********************* End Integral *********************/

    /********************** Integral ***********************/

    /* Allocation */
    cases = 6;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
```

```
1180    eps_r = calloc(cases, sizeof(pi_base_t));
1181    eps_a_corr = calloc(cases, sizeof(pi_base_t));
1182    eps_r_corr = calloc(cases, sizeof(pi_base_t));
1183    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
1184
1185    /* End Allocation */
1186
1187    /* Parallel Arrays */
1188    rules[0] = 5;
1189    fcn_eval_limit[0] = 4000000;
1190    eps_a[0] = 1.0E-06;
1191    eps_r[0] = 1.0E-06;
1192    eps_a_corr[0] = 100;
1193    eps_r_corr[0] = 100;
1194    fcn_eval_limit_corr[0] = 100;
1195
1196    rules[1] = 6;
1197    fcn_eval_limit[1] = 4000000;
1198    eps_a[1] = 1.0E-06;
1199    eps_r[1] = 1.0E-06;
1200    eps_a_corr[1] = 100;
1201    eps_r_corr[1] = 100;
1202    fcn_eval_limit_corr[1] = 100;
1203
1204    rules[2] = 7;
1205    fcn_eval_limit[2] = 4000000;
1206    eps_a[2] = 1.0E-06;
1207    eps_r[2] = 1.0E-06;
1208    eps_a_corr[2] = 100;
1209    eps_r_corr[2] = 100;
1210    fcn_eval_limit_corr[2] = 100;
1211
1212    rules[3] = 8;
1213    fcn_eval_limit[3] = 4000000;
1214    eps_a[3] = 1.0E-06;
1215    eps_r[3] = 1.0E-06;
1216    eps_a_corr[3] = 100;
1217    eps_r_corr[3] = 100;
1218    fcn_eval_limit_corr[3] = 100;
1219
1220    rules[4] = 9;
1221    fcn_eval_limit[4] = 4000000;
1222    eps_a[4] = 1.0E-06;
1223    eps_r[4] = 1.0E-06;
1224    eps_a_corr[4] = 100;
1225    eps_r_corr[4] = 100;
1226    fcn_eval_limit_corr[4] = 100;
1227
1228    rules[5] = 10;
1229    fcn_eval_limit[5] = 4000000;
1230    eps_a[5] = 1.0E-06;
1231    eps_r[5] = 1.0E-06;
1232    eps_a_corr[5] = 100;
1233    eps_r_corr[5] = 100;
1234    fcn_eval_limit_corr[5] = 100;
1235
1236    /* End Parallel Arrays */
1237
1238    /* Fill Struct */
```

```
    structArray[13].name = "fcn14";
    structArray[13].fcn = &fcn14;
    structArray[13].dim = 1;
    structArray[13].rules = rules;
    structArray[13].bounds[0] = 0.0;
    structArray[13].bounds[1] = 1.0;
    structArray[13].cases = cases;
    structArray[13].fcn_eval_limit = fcn_eval_limit;
    structArray[13].eps_a = eps_a;
    structArray[13].eps_r = eps_r;
    structArray[13].eps_a_corr = eps_a_corr;
    structArray[13].eps_r_corr = eps_r_corr;
    structArray[13].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[13].actual = 1 - (1 / pi_exp(25));

    /* End Fill Struct */

    /********************* End Integral ********************/

    /********************** Integral **********************/

    /* Allocation */
    cases = 6;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
    eps_r = calloc(cases, sizeof(pi_base_t));
    eps_a_corr = calloc(cases, sizeof(pi_base_t));
    eps_r_corr = calloc(cases, sizeof(pi_base_t));
    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));

    /* End Allocation */

    /* Parallel Arrays */
    rules[0] = 5;
    fcn_eval_limit[0] = 4000000;
    eps_a[0] = 1.0E-06;
    eps_r[0] = 1.0E-06;
    eps_a_corr[0] = 100;
    eps_r_corr[0] = 100;
    fcn_eval_limit_corr[0] = 100;

    rules[1] = 6;
    fcn_eval_limit[1] = 4000000;
    eps_a[1] = 1.0E-06;
    eps_r[1] = 1.0E-06;
    eps_a_corr[1] = 100;
    eps_r_corr[1] = 100;
    fcn_eval_limit_corr[1] = 100;

    rules[2] = 7;
    fcn_eval_limit[2] = 4000000;
    eps_a[2] = 1.0E-06;
    eps_r[2] = 1.0E-06;
    eps_a_corr[2] = 100;
    eps_r_corr[2] = 100;
    fcn_eval_limit_corr[2] = 100;

    rules[3] = 8;
```

```
1298    fcn_eval_limit[3] = 4000000;
1299    eps_a[3] = 1.0E-06;
1300    eps_r[3] = 1.0E-06;
1301    eps_a_corr[3] = 100;
1302    eps_r_corr[3] = 100;
1303    fcn_eval_limit_corr[3] = 100;
1304
1305    rules[4] = 9;
1306    fcn_eval_limit[4] = 4000000;
1307    eps_a[4] = 1.0E-06;
1308    eps_r[4] = 1.0E-06;
1309    eps_a_corr[4] = 100;
1310    eps_r_corr[4] = 100;
1311    fcn_eval_limit_corr[4] = 100;
1312
1313    rules[5] = 10;
1314    fcn_eval_limit[5] = 4000000;
1315    eps_a[5] = 1.0E-06;
1316    eps_r[5] = 1.0E-06;
1317    eps_a_corr[5] = 100;
1318    eps_r_corr[5] = 100;
1319    fcn_eval_limit_corr[5] = 100;
1320
1321    /* End Parallel Arrays */
1322
1323    /* Fill Struct */
1324    structArray[14].name = "fcn15";
1325    structArray[14].fcn = &fcn15;
1326    structArray[14].dim = 1;
1327    structArray[14].rules = rules;
1328    structArray[14].bounds[0] = 0.0;
1329    structArray[14].bounds[1] = 10.0;
1330    structArray[14].cases = cases;
1331    structArray[14].fcn_eval_limit = fcn_eval_limit;
1332    structArray[14].eps_a = eps_a;
1333    structArray[14].eps_r = eps_r;
1334    structArray[14].eps_a_corr = eps_a_corr;
1335    structArray[14].eps_r_corr = eps_r_corr;
1336    structArray[14].fcn_eval_limit_corr = fcn_eval_limit_corr;
1337    structArray[14].actual = pi_atan(500) / PI_PI;
1338
1339    /* End Fill Struct */
1340
1341    /********************* End Integral ********************/
1342
1343    /********************* Integral ********************/
1344
1345    /* Allocation */
1346    cases = 6;
1347    rules = calloc(cases, sizeof(int));
1348    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
1349    eps_a = calloc(cases, sizeof(pi_base_t));
1350    eps_r = calloc(cases, sizeof(pi_base_t));
1351    eps_a_corr = calloc(cases, sizeof(pi_base_t));
1352    eps_r_corr = calloc(cases, sizeof(pi_base_t));
1353    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
1354
1355    /* End Allocation */
1356
```

```
1357    /* Parallel Arrays */
1358    rules[0] = 5;
1359    fcn_eval_limit[0] = 4000000;
1360    eps_a[0] = 1.0E-06;
1361    eps_r[0] = 1.0E-06;
1362    eps_a_corr[0] = 100;
1363    eps_r_corr[0] = 100;
1364    fcn_eval_limit_corr[0] = 100;
1365
1366    rules[1] = 6;
1367    fcn_eval_limit[1] = 4000000;
1368    eps_a[1] = 1.0E-06;
1369    eps_r[1] = 1.0E-06;
1370    eps_a_corr[1] = 100;
1371    eps_r_corr[1] = 100;
1372    fcn_eval_limit_corr[1] = 100;
1373
1374    rules[2] = 7;
1375    fcn_eval_limit[2] = 4000000;
1376    eps_a[2] = 1.0E-06;
1377    eps_r[2] = 1.0E-06;
1378    eps_a_corr[2] = 100;
1379    eps_r_corr[2] = 100;
1380    fcn_eval_limit_corr[2] = 100;
1381
1382    rules[3] = 8;
1383    fcn_eval_limit[3] = 4000000;
1384    eps_a[3] = 1.0E-06;
1385    eps_r[3] = 1.0E-06;
1386    eps_a_corr[3] = 100;
1387    eps_r_corr[3] = 100;
1388    fcn_eval_limit_corr[3] = 100;
1389
1390    rules[4] = 9;
1391    fcn_eval_limit[4] = 4000000;
1392    eps_a[4] = 1.0E-06;
1393    eps_r[4] = 1.0E-06;
1394    eps_a_corr[4] = 100;
1395    eps_r_corr[4] = 100;
1396    fcn_eval_limit_corr[4] = 100;
1397
1398    rules[5] = 10;
1399    fcn_eval_limit[5] = 4000000;
1400    eps_a[5] = 1.0E-06;
1401    eps_r[5] = 1.0E-06;
1402    eps_a_corr[5] = 100;
1403    eps_r_corr[5] = 100;
1404    fcn_eval_limit_corr[5] = 100;
1405
1406    /* End Parallel Arrays */
1407
1408    /* Fill Struct */
1409    structArray[15].name = "fcn16";
1410    structArray[15].fcn = &fcn16;
1411    structArray[15].dim = 1;
1412    structArray[15].rules = rules;
1413    structArray[15].bounds[0] = 0.0;
1414    structArray[15].bounds[1] = 1.0;
1415    structArray[15].cases = cases;
```

```
1416     structArray[15].fcn_eval_limit = fcn_eval_limit;
1417     structArray[15].eps_a = eps_a;
1418     structArray[15].eps_r = eps_r;
1419     structArray[15].eps_a_corr = eps_a_corr;
1420     structArray[15].eps_r_corr = eps_r_corr;
1421     structArray[15].fcn_eval_limit_corr = fcn_eval_limit_corr;
1422     structArray[15].actual = 2.0 / 3.0;
1423 
1424     /* End Fill Struct */
1425 
1426     /********************** End Integral ********************/
1427 
1428     /********************** Integral ***********************/
1429 
1430     /* Allocation */
1431     cases = 6;
1432     rules = calloc(cases, sizeof(int));
1433     fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
1434     eps_a = calloc(cases, sizeof(pi_base_t));
1435     eps_r = calloc(cases, sizeof(pi_base_t));
1436     eps_a_corr = calloc(cases, sizeof(pi_base_t));
1437     eps_r_corr = calloc(cases, sizeof(pi_base_t));
1438     fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
1439 
1440     /* End Allocation */
1441 
1442     /* Parallel Arrays */
1443     rules[0] = 5;
1444     fcn_eval_limit[0] = 4000000;
1445     eps_a[0] = 1.0E-06;
1446     eps_r[0] = 1.0E-06;
1447     eps_a_corr[0] = 100;
1448     eps_r_corr[0] = 100;
1449     fcn_eval_limit_corr[0] = 100;
1450 
1451     rules[1] = 6;
1452     fcn_eval_limit[1] = 4000000;
1453     eps_a[1] = 1.0E-06;
1454     eps_r[1] = 1.0E-06;
1455     eps_a_corr[1] = 100;
1456     eps_r_corr[1] = 100;
1457     fcn_eval_limit_corr[1] = 100;
1458 
1459     rules[2] = 7;
1460     fcn_eval_limit[2] = 4000000;
1461     eps_a[2] = 1.0E-06;
1462     eps_r[2] = 1.0E-06;
1463     eps_a_corr[2] = 100;
1464     eps_r_corr[2] = 100;
1465     fcn_eval_limit_corr[2] = 100;
1466 
1467     rules[3] = 8;
1468     fcn_eval_limit[3] = 4000000;
1469     eps_a[3] = 1.0E-06;
1470     eps_r[3] = 1.0E-06;
1471     eps_a_corr[3] = 100;
1472     eps_r_corr[3] = 100;
1473     fcn_eval_limit_corr[3] = 100;
1474 
```

```
    rules[4] = 9;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    rules[5] = 10;
    fcn_eval_limit[5] = 4000000;
    eps_a[5] = 1.0E-06;
    eps_r[5] = 1.0E-06;
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[16].name = "fcn17";
    structArray[16].fcn = &fcn17;
    structArray[16].dim = 1;
    structArray[16].rules = rules;
    structArray[16].bounds[0] = 0.0;
    structArray[16].bounds[1] = 1.0;
    structArray[16].cases = cases;
    structArray[16].fcn_eval_limit = fcn_eval_limit;
    structArray[16].eps_a = eps_a;
    structArray[16].eps_r = eps_r;
    structArray[16].eps_a_corr = eps_a_corr;
    structArray[16].eps_r_corr = eps_r_corr;
    structArray[16].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[16].actual = 2 / 5;

    /* End Fill Struct */

    /********************** End Integral ********************/

    /********************** Integral ************************/

    /* Allocation */
    cases = 6;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
    eps_r = calloc(cases, sizeof(pi_base_t));
    eps_a_corr = calloc(cases, sizeof(pi_base_t));
    eps_r_corr = calloc(cases, sizeof(pi_base_t));
    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));

    /* End Allocation */

    /* Parallel Arrays */
    rules[0] = 5;
    fcn_eval_limit[0] = 4000000;
    eps_a[0] = 1.0E-06;
    eps_r[0] = 1.0E-06;
    eps_a_corr[0] = 100;
    eps_r_corr[0] = 100;
```

```
    fcn_eval_limit_corr[0] = 100;

    rules[1] = 6;
    fcn_eval_limit[1] = 4000000;
    eps_a[1] = 1.0E-06;
    eps_r[1] = 1.0E-06;
    eps_a_corr[1] = 100;
    eps_r_corr[1] = 100;
    fcn_eval_limit_corr[1] = 100;

    rules[2] = 7;
    fcn_eval_limit[2] = 4000000;
    eps_a[2] = 1.0E-06;
    eps_r[2] = 1.0E-06;
    eps_a_corr[2] = 100;
    eps_r_corr[2] = 100;
    fcn_eval_limit_corr[2] = 100;

    rules[3] = 8;
    fcn_eval_limit[3] = 4000000;
    eps_a[3] = 1.0E-06;
    eps_r[3] = 1.0E-06;
    eps_a_corr[3] = 100;
    eps_r_corr[3] = 100;
    fcn_eval_limit_corr[3] = 100;

    rules[4] = 9;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    rules[5] = 10;
    fcn_eval_limit[5] = 4000000;
    eps_a[5] = 1.0E-06;
    eps_r[5] = 1.0E-06;
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[17].name = "fcn18";
    structArray[17].fcn = &fcn18;
    structArray[17].dim = 1;
    structArray[17].rules = rules;
    structArray[17].bounds[0] = 0.0;
    structArray[17].bounds[1] = 1.0;
    structArray[17].cases = cases;
    structArray[17].fcn_eval_limit = fcn_eval_limit;
    structArray[17].eps_a = eps_a;
    structArray[17].eps_r = eps_r;
    structArray[17].eps_a_corr = eps_a_corr;
    structArray[17].eps_r_corr = eps_r_corr;
    structArray[17].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[17].actual = pi_log(2);
```

```
1593
1594    /* End Fill Struct */
1595
1596    /********************* End Integral ********************/
1597
1598    /********************** Integral ***********************/
1599
1600    /* Allocation */
1601    cases = 6;
1602    rules = calloc(cases, sizeof(int));
1603    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
1604    eps_a = calloc(cases, sizeof(pi_base_t));
1605    eps_r = calloc(cases, sizeof(pi_base_t));
1606    eps_a_corr = calloc(cases, sizeof(pi_base_t));
1607    eps_r_corr = calloc(cases, sizeof(pi_base_t));
1608    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
1609
1610    /* End Allocation */
1611
1612    /* Parallel Arrays */
1613    rules[0] = 5;
1614    fcn_eval_limit[0] = 4000000;
1615    eps_a[0] = 1.0E-06;
1616    eps_r[0] = 1.0E-06;
1617    eps_a_corr[0] = 100;
1618    eps_r_corr[0] = 100;
1619    fcn_eval_limit_corr[0] = 100;
1620
1621    rules[1] = 6;
1622    fcn_eval_limit[1] = 4000000;
1623    eps_a[1] = 1.0E-06;
1624    eps_r[1] = 1.0E-06;
1625    eps_a_corr[1] = 100;
1626    eps_r_corr[1] = 100;
1627    fcn_eval_limit_corr[1] = 100;
1628
1629    rules[2] = 7;
1630    fcn_eval_limit[2] = 4000000;
1631    eps_a[2] = 1.0E-06;
1632    eps_r[2] = 1.0E-06;
1633    eps_a_corr[2] = 100;
1634    eps_r_corr[2] = 100;
1635    fcn_eval_limit_corr[2] = 100;
1636
1637    rules[3] = 8;
1638    fcn_eval_limit[3] = 4000000;
1639    eps_a[3] = 1.0E-06;
1640    eps_r[3] = 1.0E-06;
1641    eps_a_corr[3] = 100;
1642    eps_r_corr[3] = 100;
1643    fcn_eval_limit_corr[3] = 100;
1644
1645    rules[4] = 9;
1646    fcn_eval_limit[4] = 4000000;
1647    eps_a[4] = 1.0E-06;
1648    eps_r[4] = 1.0E-06;
1649    eps_a_corr[4] = 100;
1650    eps_r_corr[4] = 100;
1651    fcn_eval_limit_corr[4] = 100;
```

```
1652
1653        rules[5] = 10;
1654        fcn_eval_limit[5] = 4000000;
1655        eps_a[5] = 1.0E-06;
1656        eps_r[5] = 1.0E-06;
1657        eps_a_corr[5] = 100;
1658        eps_r_corr[5] = 100;
1659        fcn_eval_limit_corr[5] = 100;
1660
1661        /* End Parallel Arrays */
1662
1663        /* Fill Struct */
1664        structArray[18].name = "fcn19";
1665        structArray[18].fcn = &fcn19;
1666        structArray[18].dim = 1;
1667        structArray[18].rules = rules;
1668        structArray[18].bounds[0] = 0.0;
1669        structArray[18].bounds[1] = 1.0;
1670        structArray[18].cases = cases;
1671        structArray[18].fcn_eval_limit = fcn_eval_limit;
1672        structArray[18].eps_a = eps_a;
1673        structArray[18].eps_r = eps_r;
1674        structArray[18].eps_a_corr = eps_a_corr;
1675        structArray[18].eps_r_corr = eps_r_corr;
1676        structArray[18].fcn_eval_limit_corr = fcn_eval_limit_corr;
1677        structArray[18].actual = (PI_PI + 2 * pi_atanh(1.0 / pi_sqrt(2))) / (4 * pi_sqrt
        (2));
1678
1679        /* End Fill Struct */
1680
1681        /********************** End Integral ********************/
1682
1683        /********************** Integral ************************/
1684
1685        /* Allocation */
1686        cases = 6;
1687        rules = calloc(cases, sizeof(int));
1688        fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
1689        eps_a = calloc(cases, sizeof(pi_base_t));
1690        eps_r = calloc(cases, sizeof(pi_base_t));
1691        eps_a_corr = calloc(cases, sizeof(pi_base_t));
1692        eps_r_corr = calloc(cases, sizeof(pi_base_t));
1693        fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
1694
1695        /* End Allocation */
1696
1697        /* Parallel Arrays */
1698        rules[0] = 5;
1699        fcn_eval_limit[0] = 4000000;
1700        eps_a[0] = 1.0E-06;
1701        eps_r[0] = 1.0E-06;
1702        eps_a_corr[0] = 100;
1703        eps_r_corr[0] = 100;
1704        fcn_eval_limit_corr[0] = 100;
1705
1706        rules[1] = 6;
1707        fcn_eval_limit[1] = 4000000;
1708        eps_a[1] = 1.0E-06;
1709        eps_r[1] = 1.0E-06;
```

```
    eps_a_corr[1] = 100;
    eps_r_corr[1] = 100;
    fcn_eval_limit_corr[1] = 100;

    rules[2] = 7;
    fcn_eval_limit[2] = 4000000;
    eps_a[2] = 1.0E-06;
    eps_r[2] = 1.0E-06;
    eps_a_corr[2] = 100;
    eps_r_corr[2] = 100;
    fcn_eval_limit_corr[2] = 100;

    rules[3] = 8;
    fcn_eval_limit[3] = 4000000;
    eps_a[3] = 1.0E-06;
    eps_r[3] = 1.0E-06;
    eps_a_corr[3] = 100;
    eps_r_corr[3] = 100;
    fcn_eval_limit_corr[3] = 100;

    rules[4] = 9;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    rules[5] = 10;
    fcn_eval_limit[5] = 4000000;
    eps_a[5] = 1.0E-06;
    eps_r[5] = 1.0E-06;
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[19].name = "fcn20";
    structArray[19].fcn = &fcn20;
    structArray[19].dim = 1;
    structArray[19].rules = rules;
    structArray[19].bounds[0] = 0.0;
    structArray[19].bounds[1] = 1.0;
    structArray[19].cases = cases;
    structArray[19].fcn_eval_limit = fcn_eval_limit;
    structArray[19].eps_a = eps_a;
    structArray[19].eps_r = eps_r;
    structArray[19].eps_a_corr = eps_a_corr;
    structArray[19].eps_r_corr = eps_r_corr;
    structArray[19].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[19].actual = 1 + pi_log(2) - pi_log(1 + M_E);

    /* End Fill Struct */

    /********************* End Integral ********************/

    /********************** Integral **********************/
```

```
1769
1770    /* Allocation */
1771    cases = 6;
1772    rules = calloc(cases, sizeof(int));
1773    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
1774    eps_a = calloc(cases, sizeof(pi_base_t));
1775    eps_r = calloc(cases, sizeof(pi_base_t));
1776    eps_a_corr = calloc(cases, sizeof(pi_base_t));
1777    eps_r_corr = calloc(cases, sizeof(pi_base_t));
1778    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
1779
1780    /* End Allocation */
1781
1782    /* Parallel Arrays */
1783    rules[0] = 5;
1784    fcn_eval_limit[0] = 4000000;
1785    eps_a[0] = 1.0E-06;
1786    eps_r[0] = 1.0E-06;
1787    eps_a_corr[0] = 100;
1788    eps_r_corr[0] = 100;
1789    fcn_eval_limit_corr[0] = 100;
1790
1791    rules[1] = 6;
1792    fcn_eval_limit[1] = 4000000;
1793    eps_a[1] = 1.0E-06;
1794    eps_r[1] = 1.0E-06;
1795    eps_a_corr[1] = 100;
1796    eps_r_corr[1] = 100;
1797    fcn_eval_limit_corr[1] = 100;
1798
1799    rules[2] = 7;
1800    fcn_eval_limit[2] = 4000000;
1801    eps_a[2] = 1.0E-06;
1802    eps_r[2] = 1.0E-06;
1803    eps_a_corr[2] = 100;
1804    eps_r_corr[2] = 100;
1805    fcn_eval_limit_corr[2] = 100;
1806
1807    rules[3] = 8;
1808    fcn_eval_limit[3] = 4000000;
1809    eps_a[3] = 1.0E-06;
1810    eps_r[3] = 1.0E-06;
1811    eps_a_corr[3] = 100;
1812    eps_r_corr[3] = 100;
1813    fcn_eval_limit_corr[3] = 100;
1814
1815    rules[4] = 9;
1816    fcn_eval_limit[4] = 4000000;
1817    eps_a[4] = 1.0E-06;
1818    eps_r[4] = 1.0E-06;
1819    eps_a_corr[4] = 100;
1820    eps_r_corr[4] = 100;
1821    fcn_eval_limit_corr[4] = 100;
1822
1823    rules[5] = 10;
1824    fcn_eval_limit[5] = 4000000;
1825    eps_a[5] = 1.0E-06;
1826    eps_r[5] = 1.0E-06;
1827    eps_a_corr[5] = 100;
```

```
1828    eps_r_corr[5] = 100;
1829    fcn_eval_limit_corr[5] = 100;
1830
1831    /* End Parallel Arrays */
1832
1833    /* Fill Struct */
1834    structArray[20].name = "fcn21";
1835    structArray[20].fcn = &fcn21;
1836    structArray[20].dim = 1;
1837    structArray[20].rules = rules;
1838    structArray[20].bounds[0] = 0.0;
1839    structArray[20].bounds[1] = 1.0;
1840    structArray[20].cases = cases;
1841    structArray[20].fcn_eval_limit = fcn_eval_limit;
1842    structArray[20].eps_a = eps_a;
1843    structArray[20].eps_r = eps_r;
1844    structArray[20].eps_a_corr = eps_a_corr;
1845    structArray[20].eps_r_corr = eps_r_corr;
1846    structArray[20].fcn_eval_limit_corr = fcn_eval_limit_corr;
1847
1848    pi_base_t   spenceVal = 0.40875428734889626903318497615298498;
1849    structArray[20].actual = -spenceVal - 1 + pi_pow(PI_PI, 2.0) / 6.0 + pi_log(M_E
      - 1.0);
1850
1851    /* End Fill Struct */
1852
1853    /********************** End Integral **********************/
1854
1855    /************************ Integral ************************/
1856    cases = 6;
1857    rules = calloc(cases, sizeof(int));
1858    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
1859    eps_a = calloc(cases, sizeof(pi_base_t));
1860    eps_r = calloc(cases, sizeof(pi_base_t));
1861    eps_a_corr = calloc(cases, sizeof(pi_base_t));
1862    eps_r_corr = calloc(cases, sizeof(pi_base_t));
1863    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
1864
1865    /* End Allocation */
1866
1867    /* Parallel Arrays */
1868    rules[0] = 5;
1869    fcn_eval_limit[0] = 4000000;
1870    eps_a[0] = 1.0E-06;
1871    eps_r[0] = 1.0E-06;
1872    eps_a_corr[0] = 100;
1873    eps_r_corr[0] = 100;
1874    fcn_eval_limit_corr[0] = 100;
1875
1876    rules[1] = 6;
1877    fcn_eval_limit[1] = 4000000;
1878    eps_a[1] = 1.0E-06;
1879    eps_r[1] = 1.0E-06;
1880    eps_a_corr[1] = 100;
1881    eps_r_corr[1] = 100;
1882    fcn_eval_limit_corr[1] = 100;
1883
1884    rules[2] = 7;
1885    fcn_eval_limit[2] = 4000000;
```

```
    eps_a[2] = 1.0E-06;
    eps_r[2] = 1.0E-06;
    eps_a_corr[2] = 100;
    eps_r_corr[2] = 100;
    fcn_eval_limit_corr[2] = 100;

    rules[3] = 8;
    fcn_eval_limit[3] = 4000000;
    eps_a[3] = 1.0E-06;
    eps_r[3] = 1.0E-06;
    eps_a_corr[3] = 100;
    eps_r_corr[3] = 100;
    fcn_eval_limit_corr[3] = 100;

    rules[4] = 9;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    rules[5] = 10;
    fcn_eval_limit[5] = 4000000;
    eps_a[5] = 1.0E-06;
    eps_r[5] = 1.0E-06;
    eps_a_corr[5] = 100;
    eps_r_corr[5] = 100;
    fcn_eval_limit_corr[5] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[21].name = "fcn22";
    structArray[21].fcn = &fcn22;
    structArray[21].dim = 1;
    structArray[21].rules = rules;
    structArray[21].bounds[0] = 0.0;
    structArray[21].bounds[1] = 1.0;
    structArray[21].cases = cases;
    structArray[21].fcn_eval_limit = fcn_eval_limit;
    structArray[21].eps_a = eps_a;
    structArray[21].eps_r = eps_r;
    structArray[21].eps_a_corr = eps_a_corr;
    structArray[21].eps_r_corr = eps_r_corr;
    structArray[21].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[21].actual = 2.0 / pi_sqrt(3.0);

    /* End Fill Struct */

    /********************* End Integral *********************/

    /*********************** Integral ***********************/

    /* Allocation */
    cases = 5;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
```

```
1945    eps_r = calloc(cases, sizeof(pi_base_t));
1946    eps_a_corr = calloc(cases, sizeof(pi_base_t));
1947    eps_r_corr = calloc(cases, sizeof(pi_base_t));
1948    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
1949
1950    /* End Allocation */
1951
1952    /* Parallel Arrays */
1953    rules[0] = 1;
1954    fcn_eval_limit[0] = 4000000;
1955    eps_a[0] = 1.0E-06;
1956    eps_r[0] = 1.0E-06;
1957    eps_a_corr[0] = 100;
1958    eps_r_corr[0] = 100;
1959    fcn_eval_limit_corr[0] = 100;
1960
1961    rules[1] = 3;
1962    fcn_eval_limit[1] = 4000000;
1963    eps_a[1] = 1.0E-06;
1964    eps_r[1] = 1.0E-06;
1965    eps_a_corr[1] = 100;
1966    eps_r_corr[1] = 100;
1967    fcn_eval_limit_corr[1] = 100;
1968
1969    rules[2] = 4;
1970    fcn_eval_limit[2] = 4000000;
1971    eps_a[2] = 1.0E-06;
1972    eps_r[2] = 1.0E-06;
1973    eps_a_corr[2] = 100;
1974    eps_r_corr[2] = 100;
1975    fcn_eval_limit_corr[2] = 100;
1976
1977    rules[3] = 11;
1978    fcn_eval_limit[3] = 4000000;
1979    eps_a[3] = 1.0E-06;
1980    eps_r[3] = 1.0E-06;
1981    eps_a_corr[3] = 100;
1982    eps_r_corr[3] = 100;
1983    fcn_eval_limit_corr[3] = 100;
1984
1985    rules[4] = 16;
1986    fcn_eval_limit[4] = 4000000;
1987    eps_a[4] = 1.0E-06;
1988    eps_r[4] = 1.0E-06;
1989    eps_a_corr[4] = 100;
1990    eps_r_corr[4] = 100;
1991    fcn_eval_limit_corr[4] = 100;
1992
1993    /* End Parallel Arrays */
1994
1995    /* Fill Struct */
1996    structArray[22].name = "fcn23";
1997    structArray[22].fcn = &fcn23;
1998    structArray[22].dim = 2;
1999    structArray[22].rules = rules;
2000    structArray[22].bounds[0] = 0.0;
2001    structArray[22].bounds[1] = PI_PI / 2.0;
2002    structArray[22].cases = cases;
2003    structArray[22].fcn_eval_limit = fcn_eval_limit;
```

```
2004    structArray[22].eps_a = eps_a;
2005    structArray[22].eps_r = eps_r;
2006    structArray[22].eps_a_corr = eps_a_corr;
2007    structArray[22].eps_r_corr = eps_r_corr;
2008    structArray[22].fcn_eval_limit_corr = fcn_eval_limit_corr;
2009    structArray[22].actual = PI_PI / (2 * pi_sqrt(1 - pi_pow(0.5, 2)));
2010
2011    /* End Fill Struct */
2012
2013    /********************* End Integral ********************/
2014
2015    /*********************** Integral **********************/
2016
2017    /* Allocation */
2018    cases = 5;
2019    rules = calloc(cases, sizeof(int));
2020    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
2021    eps_a = calloc(cases, sizeof(pi_base_t));
2022    eps_r = calloc(cases, sizeof(pi_base_t));
2023    eps_a_corr = calloc(cases, sizeof(pi_base_t));
2024    eps_r_corr = calloc(cases, sizeof(pi_base_t));
2025    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
2026
2027    /* End Allocation */
2028
2029    /* Parallel Arrays */
2030    rules[0] = 1;
2031    fcn_eval_limit[0] = 4000000;
2032    eps_a[0] = 1.0E-06;
2033    eps_r[0] = 1.0E-06;
2034    eps_a_corr[0] = 100;
2035    eps_r_corr[0] = 100;
2036    fcn_eval_limit_corr[0] = 100;
2037
2038    rules[1] = 3;
2039    fcn_eval_limit[1] = 4000000;
2040    eps_a[1] = 1.0E-06;
2041    eps_r[1] = 1.0E-06;
2042    eps_a_corr[1] = 100;
2043    eps_r_corr[1] = 100;
2044    fcn_eval_limit_corr[1] = 100;
2045
2046    rules[2] = 4;
2047    fcn_eval_limit[2] = 4000000;
2048    eps_a[2] = 1.0E-06;
2049    eps_r[2] = 1.0E-06;
2050    eps_a_corr[2] = 100;
2051    eps_r_corr[2] = 100;
2052    fcn_eval_limit_corr[2] = 100;
2053
2054    rules[3] = 11;
2055    fcn_eval_limit[3] = 4000000;
2056    eps_a[3] = 1.0E-06;
2057    eps_r[3] = 1.0E-06;
2058    eps_a_corr[3] = 100;
2059    eps_r_corr[3] = 100;
2060    fcn_eval_limit_corr[3] = 100;
2061
2062    rules[4] = 16;
```

```
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[23].name = "fcn24";
    structArray[23].fcn = &fcn24;
    structArray[23].dim = 2;
    structArray[23].rules = rules;
    structArray[23].bounds[0] = 0.0;
    structArray[23].bounds[1] = PI_PI / 2.0;
    structArray[23].cases = cases;
    structArray[23].fcn_eval_limit = fcn_eval_limit;
    structArray[23].eps_a = eps_a;
    structArray[23].eps_r = eps_r;
    structArray[23].eps_a_corr = eps_a_corr;
    structArray[23].eps_r_corr = eps_r_corr;
    structArray[23].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[23].actual = PI_PI * 0.5 / 2;

    /* End Fill Struct */

    /********************** End Integral **********************/

    /************************ Integral ************************/

    /* Allocation */
    cases = 5;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
    eps_r = calloc(cases, sizeof(pi_base_t));
    eps_a_corr = calloc(cases, sizeof(pi_base_t));
    eps_r_corr = calloc(cases, sizeof(pi_base_t));
    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));

    /* End Allocation */

    /* Parallel Arrays */
    rules[0] = 1;
    fcn_eval_limit[0] = 4000000;
    eps_a[0] = 1.0E-06;
    eps_r[0] = 1.0E-06;
    eps_a_corr[0] = 100;
    eps_r_corr[0] = 100;
    fcn_eval_limit_corr[0] = 100;

    rules[1] = 3;
    fcn_eval_limit[1] = 4000000;
    eps_a[1] = 1.0E-06;
    eps_r[1] = 1.0E-06;
    eps_a_corr[1] = 100;
    eps_r_corr[1] = 100;
    fcn_eval_limit_corr[1] = 100;
```

```
2122
2123    rules[2] = 4;
2124    fcn_eval_limit[2] = 4000000;
2125    eps_a[2] = 1.0E-06;
2126    eps_r[2] = 1.0E-06;
2127    eps_a_corr[2] = 100;
2128    eps_r_corr[2] = 100;
2129    fcn_eval_limit_corr[2] = 100;
2130
2131    rules[3] = 11;
2132    fcn_eval_limit[3] = 4000000;
2133    eps_a[3] = 1.0E-06;
2134    eps_r[3] = 1.0E-06;
2135    eps_a_corr[3] = 100;
2136    eps_r_corr[3] = 100;
2137    fcn_eval_limit_corr[3] = 100;
2138
2139    rules[4] = 16;
2140    fcn_eval_limit[4] = 4000000;
2141    eps_a[4] = 1.0E-06;
2142    eps_r[4] = 1.0E-06;
2143    eps_a_corr[4] = 100;
2144    eps_r_corr[4] = 100;
2145    fcn_eval_limit_corr[4] = 100;
2146
2147    /* End Parallel Arrays */
2148
2149    /* Fill Struct */
2150    structArray[24].name = "fcn25";
2151    structArray[24].fcn = &fcn25;
2152    structArray[24].dim = 2;
2153    structArray[24].rules = rules;
2154    structArray[24].bounds[0] = 0.0;
2155    structArray[24].bounds[1] = 1.0;
2156    structArray[24].cases = cases;
2157    structArray[24].fcn_eval_limit = fcn_eval_limit;
2158    structArray[24].eps_a = eps_a;
2159    structArray[24].eps_r = eps_r;
2160    structArray[24].eps_a_corr = eps_a_corr;
2161    structArray[24].eps_r_corr = eps_r_corr;
2162    structArray[24].fcn_eval_limit_corr = fcn_eval_limit_corr;
2163    structArray[24].actual = PI_PI * PI_PI / 6;
2164
2165    /* End Fill Struct */
2166
2167    /********************** End Integral ********************/
2168
2169    /************************ Integral ***********************/
2170
2171    /* Allocation */
2172    cases = 5;
2173    rules = calloc(cases, sizeof(int));
2174    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
2175    eps_a = calloc(cases, sizeof(pi_base_t));
2176    eps_r = calloc(cases, sizeof(pi_base_t));
2177    eps_a_corr = calloc(cases, sizeof(pi_base_t));
2178    eps_r_corr = calloc(cases, sizeof(pi_base_t));
2179    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
2180
```

```
    /* End Allocation */

    /* Parallel Arrays */
    rules[0] = 1;
    fcn_eval_limit[0] = 4000000;
    eps_a[0] = 1.0E-06;
    eps_r[0] = 1.0E-06;
    eps_a_corr[0] = 100;
    eps_r_corr[0] = 100;
    fcn_eval_limit_corr[0] = 100;

    rules[1] = 3;
    fcn_eval_limit[1] = 4000000;
    eps_a[1] = 1.0E-06;
    eps_r[1] = 1.0E-06;
    eps_a_corr[1] = 100;
    eps_r_corr[1] = 100;
    fcn_eval_limit_corr[1] = 100;

    rules[2] = 4;
    fcn_eval_limit[2] = 4000000;
    eps_a[2] = 1.0E-06;
    eps_r[2] = 1.0E-06;
    eps_a_corr[2] = 100;
    eps_r_corr[2] = 100;
    fcn_eval_limit_corr[2] = 100;

    rules[3] = 11;
    fcn_eval_limit[3] = 4000000;
    eps_a[3] = 1.0E-06;
    eps_r[3] = 1.0E-06;
    eps_a_corr[3] = 100;
    eps_r_corr[3] = 100;
    fcn_eval_limit_corr[3] = 100;

    rules[4] = 16;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[25].name = "fcn26";
    structArray[25].fcn = &fcn26;
    structArray[25].dim = 2;
    structArray[25].rules = rules;
    structArray[25].bounds[0] = -1.0;
    structArray[25].bounds[1] = 1.0;
    structArray[25].cases = cases;
    structArray[25].fcn_eval_limit = fcn_eval_limit;
    structArray[25].eps_a = eps_a;
    structArray[25].eps_r = eps_r;
    structArray[25].eps_a_corr = eps_a_corr;
    structArray[25].eps_r_corr = eps_r_corr;
    structArray[25].fcn_eval_limit_corr = fcn_eval_limit_corr;
```

```
2240    structArray[25].actual = -16.0 / 3.0 * (pi_sqrt(2) - 2);
2241
2242    /* End Fill Struct */
2243
2244    /********************* End Integral *********************/
2245
2246    /*********************** Integral ***********************/
2247
2248    /* Allocation */
2249    cases = 5;
2250    rules = calloc(cases, sizeof(int));
2251    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
2252    eps_a = calloc(cases, sizeof(pi_base_t));
2253    eps_r = calloc(cases, sizeof(pi_base_t));
2254    eps_a_corr = calloc(cases, sizeof(pi_base_t));
2255    eps_r_corr = calloc(cases, sizeof(pi_base_t));
2256    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
2257
2258    /* End Allocation */
2259
2260    /* Parallel Arrays */
2261    rules[0] = 1;
2262    fcn_eval_limit[0] = 4000000;
2263    eps_a[0] = 1.0E-06;
2264    eps_r[0] = 1.0E-06;
2265    eps_a_corr[0] = 100;
2266    eps_r_corr[0] = 100;
2267    fcn_eval_limit_corr[0] = 100;
2268
2269    rules[1] = 3;
2270    fcn_eval_limit[1] = 4000000;
2271    eps_a[1] = 1.0E-06;
2272    eps_r[1] = 1.0E-06;
2273    eps_a_corr[1] = 100;
2274    eps_r_corr[1] = 100;
2275    fcn_eval_limit_corr[1] = 100;
2276
2277    rules[2] = 4;
2278    fcn_eval_limit[2] = 4000000;
2279    eps_a[2] = 1.0E-06;
2280    eps_r[2] = 1.0E-06;
2281    eps_a_corr[2] = 100;
2282    eps_r_corr[2] = 100;
2283    fcn_eval_limit_corr[2] = 100;
2284
2285    rules[3] = 11;
2286    fcn_eval_limit[3] = 4000000;
2287    eps_a[3] = 1.0E-06;
2288    eps_r[3] = 1.0E-06;
2289    eps_a_corr[3] = 100;
2290    eps_r_corr[3] = 100;
2291    fcn_eval_limit_corr[3] = 100;
2292
2293    rules[4] = 16;
2294    fcn_eval_limit[4] = 4000000;
2295    eps_a[4] = 1.0E-06;
2296    eps_r[4] = 1.0E-06;
2297    eps_a_corr[4] = 100;
2298    eps_r_corr[4] = 100;
```

```
    fcn_eval_limit_corr[4] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[26].name = "fcn27";
    structArray[26].fcn = &fcn27;
    structArray[26].dim = 2;
    structArray[26].rules = rules;
    structArray[26].bounds[0] = -1.0;
    structArray[26].bounds[1] = 1.0;
    structArray[26].cases = cases;
    structArray[26].fcn_eval_limit = fcn_eval_limit;
    structArray[26].eps_a = eps_a;
    structArray[26].eps_r = eps_r;
    structArray[26].eps_a_corr = eps_a_corr;
    structArray[26].eps_r_corr = eps_r_corr;
    structArray[26].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[26].actual = 2.57900755463525232772021799987;

    //(4/3) * (-1 * 4 + -1 * pi_sqrt(2) + 3 * pi_sqrt(6) );

    /* End Fill Struct */

    /********************** End Integral ********************/

    /************************ Integral ***********************/

    /* Allocation */
    cases = 5;
    rules = calloc(cases, sizeof(int));
    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
    eps_a = calloc(cases, sizeof(pi_base_t));
    eps_r = calloc(cases, sizeof(pi_base_t));
    eps_a_corr = calloc(cases, sizeof(pi_base_t));
    eps_r_corr = calloc(cases, sizeof(pi_base_t));
    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));

    /* End Allocation */

    /* Parallel Arrays */
    rules[0] = 1;
    fcn_eval_limit[0] = 4000000;
    eps_a[0] = 1.0E-06;
    eps_r[0] = 1.0E-06;
    eps_a_corr[0] = 100;
    eps_r_corr[0] = 100;
    fcn_eval_limit_corr[0] = 100;

    rules[1] = 3;
    fcn_eval_limit[1] = 4000000;
    eps_a[1] = 1.0E-06;
    eps_r[1] = 1.0E-06;
    eps_a_corr[1] = 100;
    eps_r_corr[1] = 100;
    fcn_eval_limit_corr[1] = 100;

    rules[2] = 4;
    fcn_eval_limit[2] = 4000000;
```

```
2358     eps_a[2] = 1.0E-06;
2359     eps_r[2] = 1.0E-06;
2360     eps_a_corr[2] = 100;
2361     eps_r_corr[2] = 100;
2362     fcn_eval_limit_corr[2] = 100;
2363 
2364     rules[3] = 11;
2365     fcn_eval_limit[3] = 4000000;
2366     eps_a[3] = 1.0E-06;
2367     eps_r[3] = 1.0E-06;
2368     eps_a_corr[3] = 100;
2369     eps_r_corr[3] = 100;
2370     fcn_eval_limit_corr[3] = 100;
2371 
2372     rules[4] = 16;
2373     fcn_eval_limit[4] = 4000000;
2374     eps_a[4] = 1.0E-06;
2375     eps_r[4] = 1.0E-06;
2376     eps_a_corr[4] = 100;
2377     eps_r_corr[4] = 100;
2378     fcn_eval_limit_corr[4] = 100;
2379 
2380     /* End Parallel Arrays */
2381 
2382     /* Fill Struct */
2383     structArray[27].name = "fcn28";
2384     structArray[27].fcn = &fcn28;
2385     structArray[27].dim = 2;
2386     structArray[27].rules = rules;
2387     structArray[27].bounds[0] = -1.0;
2388     structArray[27].bounds[1] = 1.0;
2389     structArray[27].cases = cases;
2390     structArray[27].fcn_eval_limit = fcn_eval_limit;
2391     structArray[27].eps_a = eps_a;
2392     structArray[27].eps_r = eps_r;
2393     structArray[27].eps_a_corr = eps_a_corr;
2394     structArray[27].eps_r_corr = eps_r_corr;
2395     structArray[27].fcn_eval_limit_corr = fcn_eval_limit_corr;
2396     structArray[27].actual = (5.0 / 3.0) + (PI_PI / 16.0);
2397 
2398     /* End Fill Struct */
2399 
2400     /********************* End Integral ********************/
2401 
2402     /*********************** Integral **********************/
2403 
2404     /* Allocation */
2405     cases = 5;
2406     rules = calloc(cases, sizeof(int));
2407     fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
2408     eps_a = calloc(cases, sizeof(pi_base_t));
2409     eps_r = calloc(cases, sizeof(pi_base_t));
2410     eps_a_corr = calloc(cases, sizeof(pi_base_t));
2411     eps_r_corr = calloc(cases, sizeof(pi_base_t));
2412     fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
2413 
2414     /* End Allocation */
2415 
2416     /* Parallel Arrays */
```

```
    rules[0] = 2;
    fcn_eval_limit[0] = 4000000;
    eps_a[0] = 1.0E-06;
    eps_r[0] = 1.0E-06;
    eps_a_corr[0] = 100;
    eps_r_corr[0] = 100;
    fcn_eval_limit_corr[0] = 100;

    rules[1] = 3;
    fcn_eval_limit[1] = 4000000;
    eps_a[1] = 1.0E-06;
    eps_r[1] = 1.0E-06;
    eps_a_corr[1] = 100;
    eps_r_corr[1] = 100;
    fcn_eval_limit_corr[1] = 100;

    rules[2] = 4;
    fcn_eval_limit[2] = 4000000;
    eps_a[2] = 1.0E-06;
    eps_r[2] = 1.0E-06;
    eps_a_corr[2] = 100;
    eps_r_corr[2] = 100;
    fcn_eval_limit_corr[2] = 100;

    rules[3] = 11;
    fcn_eval_limit[3] = 4000000;
    eps_a[3] = 1.0E-06;
    eps_r[3] = 1.0E-06;
    eps_a_corr[3] = 100;
    eps_r_corr[3] = 100;
    fcn_eval_limit_corr[3] = 100;

    rules[4] = 16;
    fcn_eval_limit[4] = 4000000;
    eps_a[4] = 1.0E-06;
    eps_r[4] = 1.0E-06;
    eps_a_corr[4] = 100;
    eps_r_corr[4] = 100;
    fcn_eval_limit_corr[4] = 100;

    /* End Parallel Arrays */

    /* Fill Struct */
    structArray[28].name = "fcn29";
    structArray[28].fcn = &fcn29;
    structArray[28].dim = 3;
    structArray[28].rules = rules;
    structArray[28].bounds[0] = 0;
    structArray[28].bounds[1] = 1.0;
    structArray[28].cases = cases;
    structArray[28].fcn_eval_limit = fcn_eval_limit;
    structArray[28].eps_a = eps_a;
    structArray[28].eps_r = eps_r;
    structArray[28].eps_a_corr = eps_a_corr;
    structArray[28].eps_r_corr = eps_r_corr;
    structArray[28].fcn_eval_limit_corr = fcn_eval_limit_corr;
    structArray[28].actual = 0.30780727205946079028109352 2935;

    /* End Fill Struct */
```

```
2476
2477    /********************* End Integral ********************/
2478
2479    /*********************** Integral **********************/
2480
2481    /* Allocation */
2482    cases = 5;
2483    rules = calloc(cases, sizeof(int));
2484    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
2485    eps_a = calloc(cases, sizeof(pi_base_t));
2486    eps_r = calloc(cases, sizeof(pi_base_t));
2487    eps_a_corr = calloc(cases, sizeof(pi_base_t));
2488    eps_r_corr = calloc(cases, sizeof(pi_base_t));
2489    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
2490
2491    /* End Allocation */
2492
2493    /* Parallel Arrays */
2494    rules[0] = 2;
2495    fcn_eval_limit[0] = 4000000;
2496    eps_a[0] = 1.0E-06;
2497    eps_r[0] = 1.0E-06;
2498    eps_a_corr[0] = 100;
2499    eps_r_corr[0] = 100;
2500    fcn_eval_limit_corr[0] = 100;
2501
2502    rules[1] = 3;
2503    fcn_eval_limit[1] = 4000000;
2504    eps_a[1] = 1.0E-06;
2505    eps_r[1] = 1.0E-06;
2506    eps_a_corr[1] = 100;
2507    eps_r_corr[1] = 100;
2508    fcn_eval_limit_corr[1] = 100;
2509
2510    rules[2] = 4;
2511    fcn_eval_limit[2] = 4000000;
2512    eps_a[2] = 1.0E-06;
2513    eps_r[2] = 1.0E-06;
2514    eps_a_corr[2] = 100;
2515    eps_r_corr[2] = 100;
2516    fcn_eval_limit_corr[2] = 100;
2517
2518    rules[3] = 11;
2519    fcn_eval_limit[3] = 4000000;
2520    eps_a[3] = 1.0E-06;
2521    eps_r[3] = 1.0E-06;
2522    eps_a_corr[3] = 100;
2523    eps_r_corr[3] = 100;
2524    fcn_eval_limit_corr[3] = 100;
2525
2526    rules[4] = 16;
2527    fcn_eval_limit[4] = 4000000;
2528    eps_a[4] = 1.0E-06;
2529    eps_r[4] = 1.0E-06;
2530    eps_a_corr[4] = 100;
2531    eps_r_corr[4] = 100;
2532    fcn_eval_limit_corr[4] = 100;
2533
2534    /* End Parallel Arrays */
```

```
2535
2536    /* Fill Struct */
2537    structArray[29].name = "fcn30";
2538    structArray[29].fcn = &fcn30;
2539    structArray[29].dim = 3;
2540    structArray[29].rules = rules;
2541    structArray[29].bounds[0] = -1.0;
2542    structArray[29].bounds[1] = 1.0;
2543    structArray[29].cases = cases;
2544    structArray[29].fcn_eval_limit = fcn_eval_limit;
2545    structArray[29].eps_a = eps_a;
2546    structArray[29].eps_r = eps_r;
2547    structArray[29].eps_a_corr = eps_a_corr;
2548    structArray[29].eps_r_corr = eps_r_corr;
2549    structArray[29].fcn_eval_limit_corr = fcn_eval_limit_corr;
2550    structArray[29].actual = 7.01851683126717;
2551
2552    /* End Fill Struct */
2553
2554    /********************* End Integral ********************/
2555
2556    /********************** Integral **********************/
2557
2558    /* Allocation */
2559    cases = 4;
2560    rules = calloc(cases, sizeof(int));
2561    fcn_eval_limit = calloc(cases, sizeof(pi_total_t));
2562    eps_a = calloc(cases, sizeof(pi_base_t));
2563    eps_r = calloc(cases, sizeof(pi_base_t));
2564    eps_a_corr = calloc(cases, sizeof(pi_base_t));
2565    eps_r_corr = calloc(cases, sizeof(pi_base_t));
2566    fcn_eval_limit_corr = calloc(cases, sizeof(pi_total_t));
2567
2568    /* End Allocation */
2569
2570    /* Parallel Arrays */
2571    rules[0] = 3;
2572    fcn_eval_limit[0] = 4000000;
2573    eps_a[0] = 1.0E-06;
2574    eps_r[0] = 1.0E-06;
2575    eps_a_corr[0] = 100;
2576    eps_r_corr[0] = 100;
2577    fcn_eval_limit_corr[0] = 100;
2578
2579    rules[1] = 4;
2580    fcn_eval_limit[1] = 4000000;
2581    eps_a[1] = 1.0E-06;
2582    eps_r[1] = 1.0E-06;
2583    eps_a_corr[1] = 100;
2584    eps_r_corr[1] = 100;
2585    fcn_eval_limit_corr[1] = 100;
2586
2587    rules[2] = 11;
2588    fcn_eval_limit[2] = 4000000;
2589    eps_a[2] = 1.0E-06;
2590    eps_r[2] = 1.0E-06;
2591    eps_a_corr[2] = 100;
2592    eps_r_corr[2] = 100;
2593    fcn_eval_limit_corr[2] = 100;
```

```
2594
2595        rules[3] = 16;
2596        fcn_eval_limit[3] = 4000000;
2597        eps_a[3] = 1.0E-06;
2598        eps_r[3] = 1.0E-06;
2599        eps_a_corr[3] = 100;
2600        eps_r_corr[3] = 100;
2601        fcn_eval_limit_corr[3] = 100;
2602
2603        /* End Parallel Arrays */
2604
2605        /* Fill Struct */
2606        structArray[30].name = "fcn31";
2607        structArray[30].fcn = &fcn31;
2608        structArray[30].dim = 5;
2609        structArray[30].rules = rules;
2610        structArray[30].bounds[0] = 0.0;
2611        structArray[30].bounds[1] = 1.0;
2612        structArray[30].cases = cases;
2613        structArray[30].fcn_eval_limit = fcn_eval_limit;
2614        structArray[30].eps_a = eps_a;
2615        structArray[30].eps_r = eps_r;
2616        structArray[30].eps_a_corr = eps_a_corr;
2617        structArray[30].eps_r_corr = eps_r_corr;
2618        structArray[30].fcn_eval_limit_corr = fcn_eval_limit_corr;
2619        structArray[30].actual = 64 / (162435 * pi_pow(PI_PI, 5));
2620
2621        /* End Fill Struct */
2622
2623        /********************* End Integral ********************/
2624        return structArray;
2625    }
```

# Appendix C

# Family Test Pack Source Files

```
/*  File: family_driver.c
 *  Author: Austin Jones
 *  Date: 4/16/2016
 */



#include "familyTest.h"
#include "family_params.c"

pi_base_t   *alpha;                     //affective paramaters for the genz
    functions
pi_base_t   *beta;                      //unaffective paramaters for the genz
    functions
int         nPE;
int         rank;

int main(int argc, char *argv[])
/*****************************************************************************
    main()
    controller which calls multitst to run the functional tests
*****************************************************************************/
{
    if (rank == 0)
    {
        timestamp();

        printf("\n");
        printf("INTEGRAND FAMILY TESTS\n");
        printf("\n");
        printf("      Loop calling MULTST, which tests one of ParInts routines \n");
        printf("      designed to estimate multidimensional integrals\n");
        printf("\n");
        printf("      The test integrands are Genz's standard set.\n");
        printf("\n");
        printf("      This code was adapted for use with ParInt \n");
        printf("      by Austin Jones");
    }

    int     passCount = 0;
    FILE    *fp;

```

```
41      //total # of test instances (1 family-rule combo is considered an instnace)
42      int     testCount = FP.nrules * FP.tstlim;
43
44      //initialize MPI with ParInt
45      if (pi_init_mpi(&argc, &argv, &rank, &nPE) != 0)
46      {
47          printf("\n\nERROR: MPI did not initialize normally. EXITING\n\n");
48          return 1;
49      }
50
51  #ifdef DEBUG
52      printf("\nNow running on node: %d\n", rank);
53      printf("\nUsing %d processors\n", nPE);
54  #endif
55
56      //store results for each rule in an array of structs, 1 per int fam
57      struct Result_Row   *rows;
58      struct Result_Row   *benchmark_rows = malloc(FP.tstlim * sizeof(struct
        Result_Row));
59      if (rank == 0)
60      {
61          fp = fopen("benchMarkLog.txt", "r");
62          if (fp == NULL)
63          {
64              printf("\n\nERROR: No benchmark log to compare to. generateBenchmarks
            must be executed before this program. EXITING\n\n");
65              return 1;
66          }
67      }
68
69      int i;
70      for (i = 0; i < FP.nrules; i++)
71      {
72          //store the results for the rule in a struct array
73          rows = multst(FP.rules[i], FP.n_samp, FP.tstlim, FP.tstfns, FP.tstmax,
74                        FP.difclt, FP.expnts, FP.n_dims[i], FP.rule_dims[i],
75                        FP.sbname, test, FP.err_tol, FP.max_eval_lim);
76
77          //get the corresponding family results from the benchmark log
78          if (rank == 0)
79          {
80              read_result(benchmark_rows, fp, FP.tstlim);
81
82              //print a more concise result summary, tracking the total pass count
83              passCount += ReportResults(rows, benchmark_rows, FP.tstlim,
84                                         FP.rules[i], FP.rule_tol[i]);
85          }
86      }
87
88      if (rank == 0)
89      {
90          printf("\n\n %d of %d tests passed", passCount, testCount);
91          fclose(fp);
92      }
93
94      free(rows->fam);
95      free(rows);
96      free(benchmark_rows);
97
```

```
    //teardown for ParInt
    pi_finalize();

    if (rank == 0)
    {
        printf("\n");
        printf("INTEGRAND FAMILY TESTS\n");
        printf("  Normal end of execution\n");
        printf("\n");

        timestamp();
    }

    return 0;
}

struct Result_Row *multst(int rule, int nsamp, int tstlim, int tstfns[],
                          int tstmax, pi_base_t difclt[], pi_base_t expnts[],
                          int ndiml, int ndims[], char *sbname, void (*subrtn) (
                              int rule, int ndim, pi_base_t a[], pi_base_t b[],
                          int maxpts, pi_ifcn_t functn, pi_base_t rel_tol,
                          pi_base_t *errest, pi_base_t *finest, int *ifail,
                          int *eval_count), pi_base_t rel_tol, int maxpts)
/*****************************************************************************
  Purpose:
    MULTST tests a multidimensional integration routine.
  Discussion:
    The routine uses the Genz test integrand functions, with
    the user selecting the particular subset of test integrands,
    the set of difficulty factors, and the spatial dimensions.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    26 May 2007
  Author:
    Original FORTRAN77 version by Alan Genz.
    C version by John Burkardt.
  Reference:
    Alan Genz,
    A Package for Testing Multiple Integration Subroutines,
    in Numerical Integration:
    Recent Developments, Software and Applications,
    edited by Patrick Keast, Graeme Fairweather,
    D Reidel, 1987, pages 337-340,
    LC: QA299.3.N38.
  Parameters:
    Input, int NSAMP, the number of samples.
    1 <= NSAMP.
    Input, int TSTLIM, the number of test integrands.
    Input, int TSTFNS[TSTLIM], the indices of the test integrands.
    Each index is between 1 and 6.
    Input, int TSTMAX, the number of difficulty levels to be tried.
    Input, pi_base_t DIFCLT[TSTMAX], difficulty levels.
    Input, pi_base_t EXPNTS[TSTMAX], the difficulty exponents.
    Input, int NDIML, the number of sets of variable sizes.
    Input, int NDIMS[NDIML], the number of variables for the integrals
    in each test.
    Input, char *SBNAME, the name of the integration
    subroutine to be tested.
```

```
    Input, external SUBRTN, the integration subroutine to be tested.
    Input, pi_base_t REL_TOL, the relative error tolerance.
    Input, int MAXPTS, the maximum number of integrand calls
    for all tests.
*****************************************************************************/
{
    //variable declarations
    //##################################################
    pi_base_t   *a;
    pi_base_t   *b;
    pi_base_t   callsa[tstlim * tstlim];
    pi_base_t   callsb[tstlim * tstlim];
    pi_base_t   concof;
    pi_base_t   dfact;
    pi_base_t   dfclt;
    int         digits;
    pi_base_t   errest;
    pi_base_t   errlog;
    pi_base_t   ersacb[tstlim * tstlim];
    pi_base_t   ersact[tstlim * tstlim];
    pi_base_t   ersdsb[tstlim * tstlim];
    pi_base_t   ersdsc[tstlim * tstlim];
    pi_base_t   ersesb[tstlim * tstlim];
    pi_base_t   ersest[tstlim * tstlim];
    pi_base_t   ersrel[tstlim * tstlim];
    pi_base_t   estlog;
    pi_base_t   exn;
    pi_base_t   expons[tstlim];
    pi_base_t   finest;
    int         idfclt[tstlim];
    int         ifail;
    int         ifails;
    int         it;
    int         itest;
    int         j;
    int         k;
    pi_base_t   medacb[tstlim];
    pi_base_t   medacb_med[3];
    pi_base_t   *medact;
    pi_base_t   medact_med[3];
    pi_base_t   medcla[tstlim];
    pi_base_t   medcla_med[3];
    pi_base_t   medclb[tstlim];
    pi_base_t   medclb_med[3];
    pi_base_t   *medcls;
    pi_base_t   medcls_med[3];
    pi_base_t   meddsb[tstlim];
    pi_base_t   meddsb_med[3];
    pi_base_t   *meddsc;
    pi_base_t   meddsc_med[3];
    pi_base_t   medesb[tstlim];
    pi_base_t   medesb_med[3];
    pi_base_t   *medest;
    pi_base_t   medest_med[3];
    pi_base_t   medrel;
    pi_base_t   *medrll;
    pi_base_t   medrll_med[3];
    int         eval_count;
    char        *name;
```

```
    int          nconf;
    int          ndim;
    int          ndimv;
    pi_base_t    qality;
    pi_base_t    *qallty;
    pi_base_t    qallty_med[3];
    pi_base_t    qualty[tstlim * tstlim];
    int          rcalsa;
    int          rcalsb;
    pi_base_t    relerr;
    pi_base_t    small;
    pi_base_t    tactrb[tstlim];
    pi_base_t    tactrb_med[3];
    pi_base_t    tactrs[tstlim];
    pi_base_t    tactrs_med[3];
    pi_base_t    tcalsa[tstlim];
    pi_base_t    tcalsa_med[3];
    pi_base_t    tcalsb[tstlim];
    pi_base_t    tcalsb_med[3];
    pi_base_t    terdsb[tstlim];
    pi_base_t    terdsb_med[3];
    pi_base_t    terdsc[tstlim];
    pi_base_t    terdsc_med[3];
    pi_base_t    testrb[tstlim];
    pi_base_t    testrb_med[3];
    pi_base_t    testrs[tstlim];
    pi_base_t    testrs_med[3];
    pi_base_t    tqualt[tstlim];
    pi_base_t    tqualt_med[3];
    pi_base_t    total;
    pi_base_t    trelib[tstlim];
    pi_base_t    trelib_med[3];
    pi_base_t    value;

    medact = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    medcls = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    meddsc = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    medest = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    medrll = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    qallty = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));

    struct Result_Row   *rows = malloc(tstlim * sizeof(struct Result_Row));

    //#################################################
    //set up conf coef for the tests,
    // and difficulty params for the integrand families
    setup(nsamp, tstlim, &concof, &nconf, &small, idfclt, tstfns, difclt,
          expons, expnts);

    /*
    *MAIN LOOP --- NDIMS
    */
    for (ndimv = 0; ndimv < ndiml; ndimv++)
    {
        ndim = ndims[ndimv];
        a = (pi_base_t *) malloc(ndim * sizeof(pi_base_t));
        alpha = (pi_base_t *) malloc(ndim * sizeof(pi_base_t));
        b = (pi_base_t *) malloc(ndim * sizeof(pi_base_t));
        beta = (pi_base_t *) malloc(ndim * sizeof(pi_base_t));
```

```
275
276         //print out pre-test info
277         if ((ndimv % 6) == 0)
278         {
279             if (rank == 0)
280             {
281                 print_init(rule, ndiml, ndims, nsamp, tstlim, maxpts, idfclt,
282                            expons, &digits, sbname, concof, rel_tol);
283             }
284         }
285
286         /*
287         *INNER LOOP #1 --- FAMS
288         */
289         for (it = 0; it < tstlim; it++)
290         {
291             itest = tstfns[it];          //integrand to test
292             exn = expnts[itest - 1];     //family-fixed exponent (e)
293             dfclt = difclt[itest - 1];   //family-fixed difficulty paramater (h)
294
295             //assign limits of integration for each dimension (hypercube)
296             for (j = 0; j < ndim; j++)
297             {
298                 a[j] = 0.0;
299             }
300
301             for (j = 0; j < ndim; j++)
302             {
303                 b[j] = 1.0;
304             }
305
306             ifails = 0;                  //initialize the # of failed integration
    attempts to 0
307             medrel = 0;
308
309             /*
310             *INNER LOOP #2 --- SAMPLES
311             */
312             for (k = 0; k < nsamp; k++)
313             {
314                 ifail = 1;
315
316                 //generate random alpha and beta values for each test sample
317                 generate_params(ndim, itest, a, b, &total, &dfact, dfclt, exn);
318
319                 //Get the exact value of the integral
320                 value = genz_integral(itest, ndim, a, b, alpha, beta);
321
322                 ///call to integration subroutine
323                 switch (itest)
324                 {
325                     case 1:
326                         (*subrtn)
327                             (
328                                 rule, ndim, a, b, maxpts, &genz_function_1, rel_tol,
    &
329                                     errest, &finest, &ifail, &eval_count
330                             );
331                         break;
```

```
332
333                     case 2:
334                         (*subrtn)
335                             (
336                                 rule, ndim, a, b, maxpts, &genz_function_2, rel_tol,
    &
337                                 errest, &finest, &ifail, &eval_count
338                             );
339                         break;
340
341                     case 3:
342                         (*subrtn)
343                             (
344                                 rule, ndim, a, b, maxpts, &genz_function_3, rel_tol,
    &
345                                 errest, &finest, &ifail, &eval_count
346                             );
347                         break;
348
349                     case 4:
350                         (*subrtn)
351                             (
352                                 rule, ndim, a, b, maxpts, &genz_function_4, rel_tol,
    &
353                                 errest, &finest, &ifail, &eval_count
354                             );
355                         break;
356
357                     case 5:
358                         (*subrtn)
359                             (
360                                 rule, ndim, a, b, maxpts, &genz_function_5, rel_tol,
    &
361                                 errest, &finest, &ifail, &eval_count
362                             );
363                         break;
364
365                     case 6:
366                         (*subrtn)
367                             (
368                                 rule, ndim, a, b, maxpts, &genz_function_6, rel_tol,
    &
369                                 errest, &finest, &ifail, &eval_count
370                             );
371                 }
372
373                 //calculate the relative error (how accurate the approximate
    integral was)
374                 relerr = r8_abs((finest - value) / value);
375
376                 //if the max eval limit was surepassed, increment ifails
377                 ifails = ifails + i4_min(ifail, 1);
378
379                 //calculations to get wrong digits, correct digits, etc out of
    ParInt's results.
380                 relerr = r8_max(r8_min(1.0, relerr), small);
381                 errlog = r8_max(0.0, -log10(relerr));
382                 errest = r8_max(r8_min(1.0, errest), small);
383                 estlog = r8_max(0.0, -log10(errest));
```

```
384
385             //calculated values displayed in table
386             meddsc[k] = r8_max(0.0, estlog - errlog);   //wrong digits actual
387             medest[k] = estlog;     //correct digits estimated
388             medact[k] = errlog;     //correct digits actual
389             medcls[k] = eval_count;
390             if (relerr <= errest)   //TRUE if the result is reliable
391             {
392                 medrel = medrel + 1;
393             }
394         }   //END SAMPLES LOOP
395
396         /*
397         *   DIM-FAM SPECIFIC RESULTS
398         *   (each row in the results tables)
399         */
400
401         //statistical calculatations for confidence intervals
402         r8vec_median_estimate(nsamp, medest, medest_med);   //est. cor. digits
403         r8vec_median_estimate(nsamp, medact, medact_med);   //act. cor. digits
404         r8vec_median_estimate(nsamp, medcls, medcls_med);   //function evals
405         r8vec_median_estimate(nsamp, meddsc, meddsc_med);   //wrong digits
406
407         //median reliability across the samples for a family
408         medrel = medrel / (pi_base_t) (nsamp);
409
410         //calculate quality value
411         qality = 0.0;
412         if (medcls_med[0] != 0.0)
413         {
414             qality = (medact_med[0] + 1.0) * (medest_med[0] + 1.0 - meddsc_med
    [0]) / log(medcls_med[0]);
415         }
416
417         /*
418         *   fill in DIMENSION SPECIFIC MEDIANS
419         *   (dim-specific medians at the end of each table)
420         */
421         trelib[it] = medrel;
422         tqualt[it] = qality;
423         tactrs[it] = medact_med[1];
424         testrs[it] = medest_med[1];
425         terdsc[it] = meddsc_med[1];
426         tcalsa[it] = medcls_med[1];
427         tcalsb[it] = medcls_med[2];
428         tactrb[it] = medact_med[2];
429         testrb[it] = medest_med[2];
430         terdsb[it] = meddsc_med[2];
431
432         //SPECIAL CASE: of ndim = 1, the result struct must be filled in here
433         if (ndiml == 1)
434         {
435             rows[it].rule = rule;
436             rows[it].fam = genz_name(it + 1);
437             rows[it].reliability = medrel;
438             rows[it].int_evals[0] = medcls_med[1];
439             rows[it].int_evals[1] = medcls_med[2];
440             rows[it].quality = qality;
441         }
```

```
442
443             /*
444             *   fill in FAMILY-WIDE CONFIDENCE INTERVALS
445             *   (used for final summary of each family)
446             */
447
448             //fill in arrays for the corresponding dimension and family
449             ersrel[itest - 1 + ndimv * tstlim] = medrel;
450             qualty[itest - 1 + ndimv * tstlim] = qality;
451             ersest[itest - 1 + ndimv * tstlim] = medest_med[1];
452             ersact[itest - 1 + ndimv * tstlim] = medact_med[1];
453             ersdsc[itest - 1 + ndimv * tstlim] = meddsc_med[1];
454             ersesb[itest - 1 + ndimv * tstlim] = medest_med[2];
455             ersacb[itest - 1 + ndimv * tstlim] = medact_med[2];
456             ersdsb[itest - 1 + ndimv * tstlim] = meddsc_med[2];
457             callsa[itest - 1 + ndimv * tstlim] = medcls_med[1];
458             callsb[itest - 1 + ndimv * tstlim] = medcls_med[2];
459
460             //PRINT A DIM-FAM TABLE ROW
461             rcalsa = (int) medcls_med[1];
462             rcalsb = (int) medcls_med[2];
463             name = genz_name(itest);
464             if (rank == 0)
465             {
466                 printf("%4d  %14s  "PI_F(4, 2) PI_F(5, 2) PI_F(5, 2) PI_F(5, 2
467                         ) PI_F(5, 3) PI_F(4, 2) PI_F(4, 2
468                             ) "  %7d  %7d  "PI_F(6, 3) "  %5d\n", ndim, name,
469                                 medest_med[1], medest_med[2], medact_med[1],
470                                 medact_med[2], medrel, meddsc_med[1],
471                                 meddsc_med[2], rcalsa, rcalsb, qality, ifails);
472             }
473
474             free(name);
475         }   //END FAMS LOOP
476
477         //calculates the results across all dimensions tested
478         r8vec_median_estimate(tstlim, tactrs, tactrs_med);
479         r8vec_median_estimate(tstlim, trelib, trelib_med);
480         r8vec_median_estimate(tstlim, testrs, testrs_med);
481         r8vec_median_estimate(tstlim, terdsc, terdsc_med);
482         r8vec_median_estimate(tstlim, tactrb, tactrb_med);
483         r8vec_median_estimate(tstlim, testrb, testrb_med);
484         r8vec_median_estimate(tstlim, terdsb, terdsb_med);
485         r8vec_median_estimate(tstlim, tqualt, tqualt_med);
486         r8vec_median_estimate(tstlim, tcalsa, tcalsa_med);
487         r8vec_median_estimate(tstlim, tcalsb, tcalsb_med);
488
489         //PRINT MEDIANS FOR A DIM
490         rcalsa = (int) tcalsa_med[0];
491         rcalsb = (int) tcalsb_med[0];
492         if (rank == 0)
493         {
494             printf("-------------------------------------------------------
495             ------------------------------------------\n");
496             printf("%4d   Medians       "PI_F(4, 2) PI_F(5, 2) PI_F(5, 2) PI_F(
497                     5, 2) PI_F(5, 3) PI_F(4, 2) PI_F(4, 2) "%7d  %7d  "PI_F(
498                     6, 3) "\n", ndim, testrs_med[0], testrb_med[0],
499                 testrs_med[0], tactrb_med[0], trelib_med[0], terdsc_med[0],
500                 terdsb_med[0], rcalsa, rcalsb, tqualt_med[0]);
```

```
501             printf("\n");
502         }
503 
504         free(a);
505         free(alpha);
506         free(b);
507         free(beta);
508     }       //END NDIMS LOOP
509 
510     if (1 < ndiml)
511     {
512         if (rank == 0)
513         {
514             printf("-------------------------------------------------------------
515             ------------------------------------------\n");
516             printf("\n      %s Rule %d Results: \n\n", sbname, rule);
517             printf("-------------------------------------------------------------
518             ------------------------------------------\n");
519 
520             printf("\n\n");
521 
522             //print header for the tables to follow
523             printf("        Integrand       Correct digits            Relia-  Wrong
        Integrand Evals  Quality    \n");
524             printf("        Family          Estimated   Actual        bility  Digits
                                \n\n");
525         }
526 
527         for (it = 0; it < tstlim; it++)
528         {
529             itest = tstfns[it];
530 
531             for (j = 0; j < ndiml; j++)
532             {
533                 /* pull out familiy specific test-wide results (1 array per family
    containing
534                  arrays of results for each dimension tested)*/
535                 medact[j] = ersact[itest - 1 + j * tstlim];
536                 medest[j] = ersest[itest - 1 + j * tstlim];
537                 meddsc[j] = ersdsc[itest - 1 + j * tstlim];
538                 medacb[j] = ersacb[itest - 1 + j * tstlim];
539                 medesb[j] = ersesb[itest - 1 + j * tstlim];
540                 meddsb[j] = ersdsb[itest - 1 + j * tstlim];
541                 medrll[j] = ersrel[itest - 1 + j * tstlim];
542                 qallty[j] = qualty[itest - 1 + j * tstlim];
543                 medcla[j] = callsa[itest - 1 + j * tstlim];
544                 medclb[j] = callsb[itest - 1 + j * tstlim];
545             }
546 
547             //calculate the median values for the family, regardless of dimension
548             r8vec_median_estimate(ndiml, medrll, medrll_med);
549             r8vec_median_estimate(ndiml, medact, medact_med);
550             r8vec_median_estimate(ndiml, medest, medest_med);
551             r8vec_median_estimate(ndiml, meddsc, meddsc_med);
552             r8vec_median_estimate(ndiml, medacb, medacb_med);
553             r8vec_median_estimate(ndiml, medesb, medesb_med);
554             r8vec_median_estimate(ndiml, meddsb, meddsb_med);
555             r8vec_median_estimate(ndiml, qallty, qallty_med);
556             r8vec_median_estimate(ndiml, medcla, medcla_med);
```

```
557             r8vec_median_estimate(ndiml, medclb, medclb_med);
558
559             //fill in the struct row to return for summarizing results
560             rows[it].rule = rule;
561             rows[it].fam = genz_name(it + 1);
562             rows[it].reliability = medrll_med[0];
563             rows[it].int_evals[0] = rcalsa;
564             rows[it].int_evals[1] = rcalsb;
565             rows[it].quality = qallty_med[0];
566
567             //PRINT FAMILY-WIDE CONF. INT. MEDIANS
568             rcalsa = (int) medcla_med[0];
569             rcalsb = (int) medclb_med[0];
570             name = genz_name(itest);
571             if (rank == 0)
572             {
573                 printf("      %14s "PI_F(4, 2) PI_F(5, 2) PI_F(5, 2) PI_F(5, 2
574                         ) PI_F(5, 3) PI_F(4, 2) PI_F(4, 2) "%7d  %7d  "PI_F(6, 3
575                             ) "\n", name, medest_med[0], medesb_med[0],
576                                 medact_med[0], medacb_med[0], medrll_med[0],
577                                 meddsc_med[0], meddsb_med[0], rcalsa, rcalsb,
578                                 qallty_med[0]);
579             }
580
581             free(name);
582
583             //fill in family-specific medians
584             tactrs[it] = medact_med[0];
585             testrs[it] = medest_med[0];
586             terdsc[it] = meddsc_med[0];
587             tactrb[it] = medacb_med[0];
588             testrb[it] = medesb_med[0];
589             terdsb[it] = meddsb_med[0];
590             tcalsa[it] = medcla_med[0];
591             tcalsb[it] = medclb_med[0];
592             trelib[it] = medrll_med[0];
593             tqualt[it] = qallty_med[0];
594         }
595
596         //calculate medians for test-wide results, regardless of both family and
    dimension
597         r8vec_median_estimate(tstlim, tactrs, tactrs_med);
598         r8vec_median_estimate(tstlim, testrs, testrs_med);
599         r8vec_median_estimate(tstlim, terdsc, terdsc_med);
600         r8vec_median_estimate(tstlim, tactrb, tactrb_med);
601         r8vec_median_estimate(tstlim, testrb, testrb_med);
602         r8vec_median_estimate(tstlim, terdsb, terdsb_med);
603         r8vec_median_estimate(tstlim, trelib, trelib_med);
604         r8vec_median_estimate(tstlim, tqualt, tqualt_med);
605         r8vec_median_estimate(tstlim, tcalsa, tcalsa_med);
606         r8vec_median_estimate(tstlim, tcalsb, tcalsb_med);
607
608         //PRINT GLOBAL MEDIANS
609         rcalsa = (int) tcalsa_med[0];
610         rcalsb = (int) tcalsb_med[0];
611         if (rank == 0)
612         {
613             printf("-------------------------------------------------------------
614             ------------------------------------------\n");
```

```
615             printf("       Global medians "PI_F(4, 2) PI_F(5, 2) PI_F(5, 2
616                    ) PI_F(5, 2) PI_F(5, 3) PI_F(4, 2) PI_F(4, 2
617                        ) "%7d  %7d  "PI_F(6, 3) "\n", testrs_med[0],
618                            testrb_med[0], tactrs_med[0], tactrb_med[0],
619                            trelib_med[0], terdsc_med[0], terdsb_med[0], rcalsa,
620                            rcalsb, tqualt_med[0]);
621             printf("\n");
622         }
623     }
624
625     free(medact);
626     free(medcls);
627     free(meddsc);
628     free(medest);
629     free(medrll);
630     free(qallty);
631
632     return rows;
633 }
634
635 void test(int rule, int ndims, pi_base_t a[], pi_base_t b[], int maxpts,
636           pi_ifcn_t integrand, pi_base_t rel_err_tol, pi_base_t *rel_err,
637           pi_base_t *f_int_est, int *eval_lim_flag, int *eval_count)
638 /*****************************************************************************
639     test()
640     -used by multst to evaluate an integrand numerically via ParInt's API
641     -performs a single integration, called 20 times
642      per ndim + integrand family combination.
643     INPUT:
644         ndims ____________ # of dimensions to integrate over
645         a[], b[] _________ bounds of integration for each dimension
646                            (need to be of length ndims)
647         maxpts ___________ # of integrand evaluations allowed (400,000)
648         integrand ________ the function (integrand) to be integrated numerically
649                         (see genz_functions() for paramater details)
650         rel_err_tol ______ the relative error tolerance requested (1.0E-06)
651     OUTPUT:
652         rel_err __________ estimation of the relative error
653                            (estimates how accuracte the result is likely to be)
654         f_int_est ________ result of the numerical integration (this is approximate,
     not necessarily exact)
655         eval_lim_flag ____ set to 0 if maxpts # of evals is not passed, set to 1 if
    it is passed
656         eval_count _______ allows the # of integrand evaluations to be recorded and
    returned to mltst()
657                            (normally this info is printed with pi_print_results and
    a *pi_status_t)
658 *****************************************************************************/
659 {
660     int             i;
661     pi_status_t     status;
662     pi_hregion_t    *hrgn_p;
663     pi_base_t       eps_a = rel_err_tol;    //absolute error tolerance
664     pi_base_t       eps_r = rel_err_tol;    //relative error tolerance
665                                             /* --must be greater than the machine
666                                              percision of the pi_base_t being used*/
667     hrgn_p = pi_allocate_hregion(ndims);    //allocate region to integrate over
668     for (i = 0; i < ndims; i++)
669     {
```

```
670         hrgn_p->a[i] = a[i];
671         hrgn_p->b[i] = b[i];
672     }
673 
674 #ifdef DEBUG
675     if (rank == 0 && rule == 16)
676     {
677         printf("\n\n   Now calling ParInt with the following paramaters: \n");
678         printf("rule: %d  maxpts: %d  eps_a: "PI_F(1, 20) "eps_r: "PI_F(1, 20
679                ) "\n", rule, maxpts, eps_a, eps_r);
680         printf("affective paramaters:\n");
681         for (i = 0; i < ndims; i++)
682         {
683             printf("affective paramater for dim %d: "PI_F(1, 20) "\n", i,
684                    alpha[i]);
685             printf("unaffective paramater for dim %d: "PI_F(1, 20) "\n", i,
686                    beta[i]);
687             printf("integration bounds (lower, higher) for dim %d: "PI_F(1, 20
688                    ) ""PI_F(1, 20) "\n", i, a[i], b[i]);
689         }
690     }
691 #endif
692     pi_integrate(integrand, 1, rule, maxpts, eps_a, eps_r, hrgn_p, PI_RGN_HRECT,
693                  f_int_est, rel_err, &status);
694 
695 #ifdef DEBUG
696     if (rank == 0 && rule == 16)
697     {
698         printf("\n   ParInt returned with the following output: \n");
699         printf("integrand_est: "PI_F(1, 20) "\n", *f_int_est);
700         printf("relative error: "PI_F(1, 20) "\n", *rel_err);
701     }
702 #endif
703     *eval_lim_flag = status.fcn_limit_flag; //if the eval count was surpassed, set
    to 1, 0 otherwise
704     *eval_count = status.fcn_eval_count;
705 
706     pi_free_hregion(hrgn_p);                //free memory allocated for integration
    region
707 }
708 
709 void setup(int nsamp, int tstlim, pi_base_t *concof, int *nconf,
710            pi_base_t *small, int *idfclt, int *tstfns, pi_base_t *difclt,
711            pi_base_t *expons, pi_base_t *expnts)
712 /*****************************************************************************
713     setup()
714         initialize and compute confidence coefficent,
715         exponents, and difficulty levels for the integrand families
716 *****************************************************************************/
717 {
718     //Initialize and compute confidence coefficient.
719     //C > 9.5 for S > 5 (C being confidence coefficent, S being sample size)
720     int i;
721     *concof = 0.0;
722     *nconf = i4_max(1, (2 * nsamp) / 5 - 2);
723 
724     for (i = 1; i <= *nconf; i++)
725     {
726         *concof = 1.0 + (pi_base_t) (nsamp - (*nconf) + i) * (*concof) / (pi_base_t)
```

```
        ((*nconf) - i + 1);
    }

    *concof = 1.0 - (*concof) / (pi_base_t) (i4_power(2, nsamp - 1));

    *small = r8_epsilon();

    for (i = 0; i < tstlim; i++)
    {
        idfclt[i] = (int) difclt[tstfns[i] - 1];
    }

    for (i = 0; i < tstlim; i++)
    {
        expons[i] = expnts[tstfns[i] - 1];
    }
}

void print_init(int rule, int ndims, int *dims, int nsamp, int tstlim,
                int maxpts, int *idfclt, pi_base_t *expons, int *digits,
                char *sbname, pi_base_t concof, pi_base_t rel_tol)
/**********************************************************************
    print_init()
        -displays initial test information, including:
        # of samples
        difficulty levels w/ corresponding exponents
        requested # of digits
        max # of function evals
        confidence coefficent of the tests
        name of the test method
**********************************************************************/
{
    int j;
    printf("\n\n\n++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++");
    printf("\n\n     Test Paramaters for %s Rule %d: \n\n", sbname, rule);
    printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    ++++++++++++++++++++++++++++++++++++++++++++++++++\n");

    printf("\n   Showing test results with %d samples per test.\n", nsamp);

    printf("\n   Testing the following dimensions:\n\n");
    for (j = 0; j < ndims; j++)
    {
        printf("%5d", dims[j]);
    }

    printf("\n\n   Difficulty levels");
    for (j = 0; j < tstlim; j++)
    {
        printf("%6d", idfclt[j]);
    }

    printf("\n           Exponents");
    for (j = 0; j < tstlim; j++)
    {
        printf(""PI_F(3, 8) "", expons[j]);
    }

```

```
785     printf("\n");
786     *digits = (int) (-log10(rel_tol));
787     printf("\n   Requested digits = %d           Maximum values = %d\n\n",
788            *digits, maxpts);
789     printf("   Produces variable results with confidence = "PI_F(1, 20) "\n",
790            concof);
791         printf("\n-------------------------------------------------------
792     ------------------------------------------\n\n");
793 
794     //print header for the tables to follow
795     printf(" N-Dims Integrand       Correct digits            Relia-  Wrong
    Integrand Evals  Quality    Total\n");
796     printf("        Family          Estimated   Actual        bility  Digits
                            Fails\n\n");
797 }
798 
799 void generate_params(int ndim, int itest, pi_base_t *a, pi_base_t *b,
800                      pi_base_t *total, pi_base_t *dfact, pi_base_t dfclt,
801                      pi_base_t exn)
802 /*****************************************************************************
803     generate_params()
804         -called for each {ndim, rule} combination being tested, initializes the
    paramaters
805         for the nsamp tests (default 20)
806         -initializes:
807         alpha and beta (free paramater vectors)
808         a and b (region limit vectors)
809 
810         -also normalizes some of alpha, beta, a and b, for specific integrand
    families.
811         this is done using the difficulty levles and exponents
812 *****************************************************************************/
813 {
814     /*
815       Choose the integrand function parameters at random.
816       Values will be between [0,1]
817       ALPHA - the affective paramaters (don't affect all families,
818       affect difficulty of integration, so gets normed)
819       BETA - the unaffective paramaters, they should remain random and unmodified
820     */
821     int seed = 123456;                      //same seed used for all tests for
    reproducability
822     int n,
823         j;
824 
825     for (n = 0; n < ndim; n++)
826     {
827         alpha[n] = genz_random(&seed);
828         beta[n] = genz_random(&seed);
829     }
830 
831     /*
832       Modify ALPHA to account for difficulty parameter.
833     */
834     *total = r8vec_sum(ndim, alpha);
835     *dfact = *total * pow(ndim, exn) / dfclt;
836     for (j = 0; j < ndim; j++)
837     {
838         alpha[j] = alpha[j] / *dfact;
```

```
839     }
840
841     /*
842       For families 1 and 3, we modify the value of B.
843       (this has seemingly no effect on the integration, it is reducing the interval
    of integration
844       as well as the difficulty, which is essentially just an equivilent
    transformation)
845     */
846     if (itest == 1 || itest == 3)
847     {
848         for (j = 0; j < ndim; j++)
849         {
850             b[j] = alpha[j];
851         }
852     }
853
854     /*
855       For test 6, we modify the value of BETA.
856       (this is because the values of beta beyond X2 are irrelevent, we only care
    about the first 2
857       so set these others to known quantities)
858     */
859     if (itest == 6)
860     {
861         for (n = 2; n < ndim; n++)
862         {
863             beta[n] = 1.0;
864         }
865     }
866 }
867
868 void read_result(struct Result_Row *rows, FILE *fp, int n_fams)
869 /*****************************************************************************
870 *****************************************************************************/
871 {
872     int         i;
873     int         rule;
874     char        *fam = (char *) malloc(14 * sizeof(char));
875     pi_base_t   rel;
876     int         evals_a;
877     int         evals_b;
878     pi_base_t   qual;
879     int         read;
880
881     for (i = 0; i < n_fams; i++)
882     {
883         read = fscanf(fp, "%d  %s  "PI_F(1, 20) "%d  %d  "PI_F(2, 20) "\n",
884                       &rule, fam, &rel, &evals_a, &evals_b, &qual);
885
886         rows[i].rule = rule;
887         rows[i].fam = fam;
888         rows[i].reliability = rel;
889         rows[i].int_evals[0] = evals_a;
890         rows[i].int_evals[1] = evals_b;
891         rows[i].quality = qual;
892     }
893 }
894
```

```
int ReportResults(struct Result_Row *rows, struct Result_Row *bench, int n_fams,
                  int rule, pi_base_t *rule_tol)
/*************************************************************************
    ReportResults()
    -given a set of row results for a given (ParInt-rule, integrand-family) set,
     determines which of the 6 tests passed or failed, sumarizing the results
*************************************************************************/
{
    int i;
    int avg_int_evals;
    int bench_int_evals;
    int passCount = 0;

    //check which tests passed
    for (i = 0; i < n_fams; i++)
    {
        avg_int_evals = (rows[i].int_evals[0] + rows[i].int_evals[1]) / 2;
        bench_int_evals = (bench[i].int_evals[0] + bench[i].int_evals[1]) / 2;
        rows[i].pass = 1;                    //assumed passed until a check is failed
        if (rows[i].reliability < bench[i].reliability - rule_tol[0])
        {
            rows[i].pass = 0;
        }
        else if (rows[i].quality < bench[i].quality - rule_tol[1])
        {
            rows[i].pass = 0;
        }
        else if (avg_int_evals > bench_int_evals + rule_tol[2])
        {
            rows[i].pass = 0;
        }
    }

    //print the rule and family of failed tests, count # of failures
    for (i = 0; i < n_fams; i++)
    {
        if (rows[i].pass)
        {
            passCount++;
            if (rank == 0)
            {
                printf("\nrule # %d on family %s ... OK\n", rows[i].rule,
                       rows[i].fam);
            }
        }
        else
        {
            if (rank == 0)
            {
                printf("\nTest of rule # %d on family %s failed to meet minimum
   benchmarks. FAILED\n",
                   rows[i].rule, rows[i].fam);
            }
        }
    }

    return passCount;
}
```

```c
/*  File: family_integrals.c
 *  Edited by Austin Jones
 *  Date: 4/16/2016
 */

#include "familyTest.h"

/*****************************************************************************/
pi_base_t genz_integral(int indx, int ndim, pi_base_t a[], pi_base_t b[],
                        pi_base_t alpha[], pi_base_t beta[])
/*****************************************************************************/

/*
  Purpose:
    GENZ_INTEGRAL computes the exact integrals of the test functions.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    26 May 2007
  Author:
    Original FORTRAN77 version by Alan Genz.
    C version by John Burkardt.
  Reference:
    Alan Genz,
    A Package for Testing Multiple Integration Subroutines,
    in Numerical Integration:
    Recent Developments, Software and Applications,
    edited by Patrick Keast, Graeme Fairweather,
    D Reidel, 1987, pages 337-340,
    LC: QA299.3.N38.
  Parameters:
    Input, int INDX, the index of the test.
    Input, int NDIM, the spatial dimension.
    Input, pi_base_t A[NDIM], B[NDIM], the lower and upper limits
    of integration.
    Input, pi_base_t ALPHA[NDIM], BETA[NDIM], parameters
    associated with the integrand function.
    Output, pi_base_t GENZ_INTEGRAL, the exact value of the integral.
*/
{
    pi_base_t       ab;
    int             *ic;
    int             isum;
    int             j;
    const pi_base_t pi = 3.14159265358979323844;
    int             rank;
    pi_base_t       s;
    pi_base_t       sgndm;
    pi_base_t       total;
    pi_base_t       value;

    /*
  Oscillatory.
*/
    if (indx == 1)
    {
        value = 0.0;

        /*
```

```
  Generate all sequences of NDIM 0's and 1's.
*/
        rank = 0;
        ic = (int *) malloc(ndim * sizeof(int));

        for (;;)
        {
            tuple_next(0, 1, ndim, &rank, ic);

            if (rank == 0)
            {
                break;
            }

            total = 2.0 * pi * beta[0];
            for (j = 0; j < ndim; j++)
            {
                if (ic[j] != 1)
                {
                    total = total + alpha[j];
                }
            }

            isum = i4vec_sum(ndim, ic);

            s = 1 + 2 * ((isum / 2) * 2 - isum);

            if ((ndim % 2) == 0)
            {
                value = value + s * pi_cos(total);
            }
            else
            {
                value = value + s * pi_sin(total);
            }
        }

        free(ic);

        if (1 < (ndim % 4))
        {
            value = -value;
        }
    }

    /*
  Product Peak.
*/
    else if (indx == 2)
    {
        value = 1.0;

        for (j = 0; j < ndim; j++)
        {
            value = value * alpha[j] * (pi_atan((1.0 - beta[j]) * alpha[j]) +
    pi_atan(+beta[j] * alpha[j]));
        }
    }

```

```
    /*
  Corner Peak.
*/
    else if (indx == 3)
    {
        value = 0.0;

        sgndm = 1.0;
        for (j = 1; j <= ndim; j++)
        {
            sgndm = -sgndm / (pi_base_t) (j);
        }

        rank = 0;
        ic = (int *) malloc(ndim * sizeof(int));

        for (;;)
        {
            tuple_next(0, 1, ndim, &rank, ic);

            if (rank == 0)
            {
                break;
            }

            total = 1.0;

            for (j = 0; j < ndim; j++)
            {
                if (ic[j] != 1)
                {
                    total = total + alpha[j];
                }
            }

            isum = i4vec_sum(ndim, ic);

            s = 1 + 2 * ((isum / 2) * 2 - isum);
            value = value + (pi_base_t) s / total;
        }

        free(ic);

        value = value * sgndm;
    }

    /*
  Gaussian.
*/
    else if (indx == 4)
    {
        value = 1.0;

        ab = pi_sqrt(2.0);
        for (j = 0; j < ndim; j++)
        {
            value = value * (pi_sqrt(pi) / alpha[j]) * (genz_phi((1.0 - beta[j]) *
   ab * alpha[j]) - genz_phi(-beta[j] * ab * alpha[j]));
        }
```

```
    }

    /*
  C0 Function.
*/
    else if (indx == 5)
    {
        value = 1.0;
        for (j = 0; j < ndim; j++)
        {
            ab = alpha[j] * beta[j];
            value = value *
                (2.0 - pi_exp(-ab) - pi_exp(ab - alpha[j])) /
                alpha[j];
        }
    }

    /*
  Discontinuous.
*/
    else if (indx == 6)
    {
        value = 1.0;
        for (j = 0; j < ndim; j++)
        {
            value = value * (pi_exp(alpha[j] * beta[j]) - 1.0) / alpha[j];
        }
    }

    return value;
}

/******************************************************************************/
```

```
/*  File: family_integrands.c
 *  Edited by Austin Jones
 *  Date: 4/16/2016
 */



#include "familyTest.h"

/******************************************************************************/
int genz_function_1(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
/******************************************************************************/

/*
  Purpose:
    GENZ_FUNCTION_1 evaluates one of the test integrand functions.
  Parameters:
    Input, int *NDIMS, the spatial dimension.
    Input, pi_base_t X[*NDIMS], the point at which the integrand
    is to be evaluated.
    Output, pi_base_t GENZ_FUNCTION, the value of the test function.
*/
{
    const pi_base_t pi = 3.14159265358979323844;
    pi_base_t       total;
```

```
    funvls[0] = 0.0;

    /*
  Oscillatory.
*/
    total = 2.0 * pi * beta[0] + r8vec_sum(*ndims, x);
    funvls[0] = pi_cos(total);
    return 0;
}

/*****************************************************************************/
int genz_function_2(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
/*****************************************************************************/

/*
   Purpose:
    GENZ_FUNCTION_2 evaluates one of the test integrand functions.
  Parameters:
    Input, int *NDIMS, the spatial dimension.
    Input, pi_base_t X[*NDIMS], the point at which the integrand
    is to be evaluated.
    Output, pi_base_t GENZ_FUNCTION, the value of the test function.
*/
{
    int         j;

    //const pi_base_t pi = 3.14159265358979323844;
    //int test;
    pi_base_t   total;
    funvls[0] = 0.0;

    /*
  Product Peak.
*/
    total = 1.0;
    for (j = 0; j < *ndims; j++)
    {
        total = total * (1.0 / pi_pow(alpha[j], 2) + pi_pow(x[j] - beta[j], 2));
    }

    funvls[0] = 1.0 / total;
    return 0;
}

/*****************************************************************************/
int genz_function_3(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
/*****************************************************************************/

/*
   Purpose:
    GENZ_FUNCTION_3 evaluates one of the test integrand functions.
  Parameters:
    Input, int *NDIMS, the spatial dimension.
    Input, pi_base_t X[*NDIMS], the point at which the integrand
    is to be evaluated.
    Output, pi_base_t GENZ_FUNCTION, the value of the test function.
*/
{
    int         j;
```

```
85
86      //const pi_base_t pi = 3.14159265358979323844;
87      //int test;
88      pi_base_t   total;
89      funvls[0] = 0.0;
90
91      /*
92    Corner Peak.
93  */
94
95      /*
96    For this case, the BETA's are used to randomly select
97    a corner for the peak.
98  */
99      total = 1.0;
100     for (j = 0; j < *ndims; j++)
101     {
102         if (beta[j] < 0.5)
103         {
104             total = total + x[j];
105         }
106         else
107         {
108             total = total + alpha[j] - x[j];
109         }
110     }
111
112     funvls[0] = 1.0 / pi_pow(total, *ndims + 1);
113     return 0;
114 }
115
116 /*****************************************************************************/
117 int genz_function_4(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
118 /*****************************************************************************/
119
120 /*
121    Purpose:
122     GENZ_FUNCTION_4 evaluates one of the test integrand functions.
123   Parameters:
124     Input, int *NDIMS, the spatial dimension.
125     Input, pi_base_t X[*NDIMS], the point at which the integrand
126     is to be evaluated.
127     Output, pi_base_t GENZ_FUNCTION, the value of the test function.
128 */
129 {
130     int         j;
131
132     //const pi_base_t pi = 3.14159265358979323844;
133     //int test;
134     pi_base_t   total;
135     funvls[0] = 0.0;
136
137     /*
138   Gaussian.
139   C math library complains about things like exp ( -700 )!
140 */
141     total = 0.0;
142     for (j = 0; j < *ndims; j++)
143     {
```

```
        total = total + pi_pow(alpha[j] * (x[j] - beta[j]), 2);
    }

    total = r8_min(total, 100.0);
    funvls[0] = pi_exp(-total);
    return 0;
}

/******************************************************************************/
int genz_function_5(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
/******************************************************************************/

/*
   Purpose:
    GENZ_FUNCTION_5 evaluates one of the test integrand functions.
  Parameters:
    Input, int *NDIMS, the spatial dimension.
    Input, pi_base_t X[*NDIMS], the point at which the integrand
    is to be evaluated.
    Output, pi_base_t GENZ_FUNCTION, the value of the test function.
*/
{
    int         j;

    //const pi_base_t pi = 3.14159265358979323844;
    //int test;
    pi_base_t   total;

    funvls[0] = 0.0;

    /*
  C0 Function.
*/
    total = 0.0;
    for (j = 0; j < *ndims; j++)
    {
        total = total + alpha[j] * r8_abs(x[j] - beta[j]);
    }

    funvls[0] = pi_exp(-total);
    return 0;
}

/******************************************************************************/
int genz_function_6(int *ndims, pi_base_t x[], int *nfcns, pi_base_t funvls[])
/******************************************************************************/

/*
   Purpose:
    GENZ_FUNCTION_6 evaluates one of the test integrand functions.
  Parameters:
    Input, int *NDIMS, the spatial dimension.
    Input, pi_base_t X[*NDIMS], the point at which the integrand
    is to be evaluated.
    Output, pi_base_t GENZ_FUNCTION, the value of the test function.
*/
{
    int         j;

```

```
    //const pi_base_t pi = 3.14159265358979323844;
    int          test;
    pi_base_t    total;
    funvls[0] = 0.0;

    /*
  Discontinuous.
*/
    test = 0;
    for (j = 0; j < *ndims; j++)
    {
        if (beta[j] < x[j])
        {
            test = 1;
            break;
        }
    }

    if (test)
    {
        funvls[0] = 0.0;
    }
    else
    {
        total = r8vec_dot_product(*ndims, alpha, x);
        funvls[0] = pi_exp(total);
    }

    return 0;
}

/*****************************************************************************/
```

```
/*
/*  File: family_params.c
 *  Author: Austin Jones
 *  Date: 4/16/2016
 */
  -holds all test-wide paramaters for the faimily tests
  -keeps generateBenchmarks and familyTest in sync so the log's
  -params match those in this struct before familyTest will run
*/
struct FamilyParams
{
    int          tstlim;
    int          tstmax;
    int          max_eval_lim;
    int          n_samp;
    pi_base_t    err_tol;
    int          nrules;
    int          maxdim;
    char         *sbname;
    int          rules[12];
    int          n_dims[12];
    int          rule_dims[12][10];
    pi_base_t    rule_tol[12][3];
    pi_base_t    difclt[6];
    pi_base_t    expnts[6];
    int          tstfns[6];
```

```
};

struct FamilyParams FP =
{
    .tstlim = 6, .tstmax = 6, .max_eval_lim = 400000, .n_samp = 20,
    .err_tol = 1.0E-06, .nrules = 11, .maxdim = 6, .sbname = "ParInt", .rules =
    { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16 }, .n_dims =
    { 1, 1, 5, 5, 1, 1, 1, 1, 1, 1, 5 }, .rule_dims = {
        { 2 }, { 3 }, { 2, 3, 4, 6, 8 }, { 2, 3, 4, 6, 8 }, { 1 }, { 1 }, { 1 },
            { 1 }, { 1 }, { 1 }, { 2, 3, 4, 6, 8 } }, .rule_tol = {
        { 0.1, 0.1, 1 }, { 0.1, 0.1, 1 }, { 0.1, 0.1, 1 }, { 0.1, 0.1, 1 }, {
                0.1, 0.1, 1 }, { 0.1, 0.1, 1 },
        { 0.1, 0.1, 1 }, { 0.1, 0.1, 1 }, { 0.1, 0.1, 1 }, { 0.1, 0.1, 1 }, {
                0.1, 0.1, 1 } }, .difclt =
    { 110.0, 600.0, 600.0, 100.0, 150.0, 100.0 }, .expnts =
    { 1.5, 2.0, 2.0, 1.0, 2.0, 2.0 }, .tstfns = { 1, 2, 3, 4, 5, 6 }
};
```

```
/*  File: benchmark_builder.c
 *  Author: Austin Jones
 *  Date: 4/16/2016
 */

#include "familyTest.h"
#include "family_params.c"

pi_base_t   *alpha;                     //affective paramaters for the genz
    functions
pi_base_t   *beta;                      //unaffective paramaters for the genz
    functions
int         nPE;
int         rank;

int main(int argc, char *argv[])
/*****************************************************************************
    main()
    controller which calls multitst to run the functional tests
*****************************************************************************/
{
    if (rank == 0)
    {
        timestamp();

        printf("\n");
        printf("INTEGRAND FAMILY TESTS\n");
        printf("\n");
        printf("      Loop calling MULTST, which tests one of ParInts routines \n");
        printf("      designed to estimate multidimensional integrals\n");
        printf("\n");
        printf("      The test integrands are Genz's standard set.\n");
        printf("\n");
        printf("      This code was adapted for use with ParInt \n");
        printf("      by Austin Jones & Lawrence Cuneaz");
    }

    FILE    *fp;

    //initialize MPI with ParInt
    pi_init_mpi(&argc, &argv, &rank, &nPE);
```

```
40
41      //store results for each rule in an array of structs, 1 per int fam
42      struct Result_Row   *rows;
43
44      //create log to write benchmark results to
45      if (rank == 0)
46      {
47          fp = fopen("benchMarkLog.txt", "w+");
48      }
49
50      int i;
51      for (i = 0; i < FP.nrules; i++)
52      {
53          //store the results for each rule and the current family in a struct array
54          rows = multst(FP.rules[i], FP.n_samp, FP.tstlim, FP.tstfns, FP.tstmax,
55                        FP.difclt, FP.expnts, FP.n_dims[i], FP.rule_dims[i],
56                        FP.sbname, test, FP.err_tol, FP.max_eval_lim);
57
58          //write the test result to the log file
59          if (rank == 0)
60          {
61              write_result(rows, fp, FP.tstlim);
62          }
63      }
64
65      if (rank == 0)
66      {
67          fclose(fp);
68      }
69
70      free(rows);
71
72      //teardown for ParInt
73      pi_finalize();
74
75      if (rank == 0)
76      {
77          printf("\n");
78          printf("INTEGRAND FAMILY TESTS\n");
79          printf("  Normal end of execution\n");
80          printf("\n");
81      }
82
83      timestamp();
84
85      return 0;
86  }
87
88  struct Result_Row *multst(int rule, int nsamp, int tstlim, int tstfns[],
89                            int tstmax, pi_base_t difclt[], pi_base_t expnts[],
90                            int ndiml, int ndims[], char *sbname, void (*subrtn) (
91                                int rule, int ndim, pi_base_t a[], pi_base_t b[],
92                            int maxpts, pi_ifcn_t functn, pi_base_t rel_tol,
93                            pi_base_t *errest, pi_base_t *finest, int *ifail,
94                            int *eval_count), pi_base_t rel_tol, int maxpts)
95  /******************************************************************************
96    Purpose:
97      MULTST tests a multidimensional integration routine.
98    Discussion:
```

```
99      The routine uses the Genz test integrand functions, with
100     the user selecting the particular subset of test integrands,
101     the set of difficulty factors, and the spatial dimensions.
102   Licensing:
103     This code is distributed under the GNU LGPL license.
104   Modified:
105     26 May 2007
106   Author:
107     Original FORTRAN77 version by Alan Genz.
108     C version by John Burkardt.
109   Reference:
110     Alan Genz,
111     A Package for Testing Multiple Integration Subroutines,
112     in Numerical Integration:
113     Recent Developments, Software and Applications,
114     edited by Patrick Keast, Graeme Fairweather,
115     D Reidel, 1987, pages 337-340,
116     LC: QA299.3.N38.
117   Parameters:
118     Input, int NSAMP, the number of samples.
119     1 <= NSAMP.
120     Input, int TSTLIM, the number of test integrands.
121     Input, int TSTFNS[TSTLIM], the indices of the test integrands.
122     Each index is between 1 and 6.
123     Input, int TSTMAX, the number of difficulty levels to be tried.
124     Input, pi_base_t DIFCLT[TSTMAX], difficulty levels.
125     Input, pi_base_t EXPNTS[TSTMAX], the difficulty exponents.
126     Input, int NDIML, the number of sets of variable sizes.
127     Input, int NDIMS[NDIML], the number of variables for the integrals
128     in each test.
129     Input, char *SBNAME, the name of the integration
130     subroutine to be tested.
131     Input, external SUBRTN, the integration subroutine to be tested.
132     Input, pi_base_t REL_TOL, the relative error tolerance.
133     Input, int MAXPTS, the maximum number of integrand calls
134     for all tests.
135 ******************************************************************************/
136 {
137     //variable declarations
138     //################################################
139     pi_base_t   *a;
140     pi_base_t   *b;
141     pi_base_t   callsa[tstlim * tstlim];
142     pi_base_t   callsb[tstlim * tstlim];
143     pi_base_t   concof;
144     pi_base_t   dfact;
145     pi_base_t   dfclt;
146     int         digits;
147     pi_base_t   errest;
148     pi_base_t   errlog;
149     pi_base_t   ersacb[tstlim * tstlim];
150     pi_base_t   ersact[tstlim * tstlim];
151     pi_base_t   ersdsb[tstlim * tstlim];
152     pi_base_t   ersdsc[tstlim * tstlim];
153     pi_base_t   ersesb[tstlim * tstlim];
154     pi_base_t   ersest[tstlim * tstlim];
155     pi_base_t   ersrel[tstlim * tstlim];
156     pi_base_t   estlog;
157     pi_base_t   exn;
```

```
158     pi_base_t   expons[tstlim];
159     pi_base_t   finest;
160     int         idfclt[tstlim];
161     int         ifail;
162     int         ifails;
163     int         it;
164     int         itest;
165     int         j;
166     int         k;
167     pi_base_t   medacb[tstlim];
168     pi_base_t   medacb_med[3];
169     pi_base_t   *medact;
170     pi_base_t   medact_med[3];
171     pi_base_t   medcla[tstlim];
172     pi_base_t   medcla_med[3];
173     pi_base_t   medclb[tstlim];
174     pi_base_t   medclb_med[3];
175     pi_base_t   *medcls;
176     pi_base_t   medcls_med[3];
177     pi_base_t   meddsb[tstlim];
178     pi_base_t   meddsb_med[3];
179     pi_base_t   *meddsc;
180     pi_base_t   meddsc_med[3];
181     pi_base_t   medesb[tstlim];
182     pi_base_t   medesb_med[3];
183     pi_base_t   *medest;
184     pi_base_t   medest_med[3];
185     pi_base_t   medrel;
186     pi_base_t   *medrll;
187     pi_base_t   medrll_med[3];
188     int         eval_count;
189     char        *name;
190     int         nconf;
191     int         ndim;
192     int         ndimv;
193     pi_base_t   qality;
194     pi_base_t   *qallty;
195     pi_base_t   qallty_med[3];
196     pi_base_t   qualty[tstlim * tstlim];
197     int         rcalsa;
198     int         rcalsb;
199     pi_base_t   relerr;
200     pi_base_t   small;
201     pi_base_t   tactrb[tstlim];
202     pi_base_t   tactrb_med[3];
203     pi_base_t   tactrs[tstlim];
204     pi_base_t   tactrs_med[3];
205     pi_base_t   tcalsa[tstlim];
206     pi_base_t   tcalsa_med[3];
207     pi_base_t   tcalsb[tstlim];
208     pi_base_t   tcalsb_med[3];
209     pi_base_t   terdsb[tstlim];
210     pi_base_t   terdsb_med[3];
211     pi_base_t   terdsc[tstlim];
212     pi_base_t   terdsc_med[3];
213     pi_base_t   testrb[tstlim];
214     pi_base_t   testrb_med[3];
215     pi_base_t   testrs[tstlim];
216     pi_base_t   testrs_med[3];
```

```
    pi_base_t   tqualt[tstlim];
    pi_base_t   tqualt_med[3];
    pi_base_t   total;
    pi_base_t   trelib[tstlim];
    pi_base_t   trelib_med[3];
    pi_base_t   value;

    medact = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    medcls = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    meddsc = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    medest = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    medrll = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));
    qallty = (pi_base_t *) malloc(nsamp * sizeof(pi_base_t));

    struct Result_Row   *rows = malloc(tstlim * sizeof(struct Result_Row));

    //################################################
    //set up conf coef for the tests,
    // and difficulty params for the integrand families
    setup(nsamp, tstlim, &concof, &nconf, &small, idfclt, tstfns, difclt,
          expons, expnts);

    /*
    *MAIN LOOP --- NDIMS
    */
    for (ndimv = 0; ndimv < ndiml; ndimv++)
    {
        ndim = ndims[ndimv];
        a = (pi_base_t *) malloc(ndim * sizeof(pi_base_t));
        alpha = (pi_base_t *) malloc(ndim * sizeof(pi_base_t));
        b = (pi_base_t *) malloc(ndim * sizeof(pi_base_t));
        beta = (pi_base_t *) malloc(ndim * sizeof(pi_base_t));

        //print out pre-test info
        if ((ndimv % 6) == 0)
        {
            if (rank == 0)
            {
                print_init(rule, ndiml, ndims, nsamp, tstlim, maxpts, idfclt,
                           expons, &digits, sbname, concof, rel_tol);
            }
        }

        /*
        *INNER LOOP #1 --- FAMS
        */
        for (it = 0; it < tstlim; it++)
        {
            itest = tstfns[it];          //integrand to test
            exn = expnts[itest - 1];     //family-fixed exponent (e)
            dfclt = difclt[itest - 1];   //family-fixed difficulty paramater (h)

            //assign limits of integration for each dimension (hypercube)
            for (j = 0; j < ndim; j++)
            {
                a[j] = 0.0;
            }

            for (j = 0; j < ndim; j++)
```

```
276             {
277                 b[j] = 1.0;
278             }
279 
280             ifails = 0;                 //initialize the # of failed integration
    attempts to 0
281             medrel = 0;
282 
283             /*
284             *INNER LOOP #2 --- SAMPLES
285             */
286             for (k = 0; k < nsamp; k++)
287             {
288                 ifail = 1;
289 
290                 //generate random alpha and beta values for each test sample
291                 generate_params(ndim, itest, a, b, &total, &dfact, dfclt, exn);
292 
293                 //Get the exact value of the integral
294                 value = genz_integral(itest, ndim, a, b, alpha, beta);
295 
296                 ///call to integration subroutine
297                 switch (itest)
298                 {
299                     case 1:
300                         (*subrtn)
301                             (
302                                 rule, ndim, a, b, maxpts, &genz_function_1, rel_tol,
    &
303                                     errest, &finest, &ifail, &eval_count
304                             );
305                         break;
306 
307                     case 2:
308                         (*subrtn)
309                             (
310                                 rule, ndim, a, b, maxpts, &genz_function_2, rel_tol,
    &
311                                     errest, &finest, &ifail, &eval_count
312                             );
313                         break;
314 
315                     case 3:
316                         (*subrtn)
317                             (
318                                 rule, ndim, a, b, maxpts, &genz_function_3, rel_tol,
    &
319                                     errest, &finest, &ifail, &eval_count
320                             );
321                         break;
322 
323                     case 4:
324                         (*subrtn)
325                             (
326                                 rule, ndim, a, b, maxpts, &genz_function_4, rel_tol,
    &
327                                     errest, &finest, &ifail, &eval_count
328                             );
329                         break;
```

```
330
331                     case 5:
332                         (*subrtn)
333                             (
334                                 rule, ndim, a, b, maxpts, &genz_function_5, rel_tol,
    &
335                                     errest, &finest, &ifail, &eval_count
336                             );
337                         break;
338
339                     case 6:
340                         (*subrtn)
341                             (
342                                 rule, ndim, a, b, maxpts, &genz_function_6, rel_tol,
    &
343                                     errest, &finest, &ifail, &eval_count
344                             );
345                 }
346
347                 //calculate the relative error (how accurate the approximate
    integral was)
348                 relerr = r8_abs((finest - value) / value);
349
350                 //if the max eval limit was surepassed, increment ifails
351                 ifails = ifails + i4_min(ifail, 1);
352
353                 //calculations to get wrong digits, correct digits, etc out of
    ParInt's results.
354                 relerr = r8_max(r8_min(1.0, relerr), small);
355                 errlog = r8_max(0.0, -log10(relerr));
356                 errest = r8_max(r8_min(1.0, errest), small);
357                 estlog = r8_max(0.0, -log10(errest));
358
359                 //calculated values displayed in table
360                 meddsc[k] = r8_max(0.0, estlog - errlog);   //wrong digits actual
361                 medest[k] = estlog;     //correct digits estimated
362                 medact[k] = errlog;     //correct digits actual
363                 medcls[k] = eval_count;
364                 if (relerr <= errest)   //TRUE if the result is reliable
365                 {
366                     medrel = medrel + 1;
367                 }
368             }   //END SAMPLES LOOP
369
370             /*
371             *   DIM-FAM SPECIFIC RESULTS
372             *   (each row in the results tables)
373             */
374
375             //statistical calculatations for confidence intervals
376             r8vec_median_estimate(nsamp, medest, medest_med);   //est. cor. digits
377             r8vec_median_estimate(nsamp, medact, medact_med);   //act. cor. digits
378             r8vec_median_estimate(nsamp, medcls, medcls_med);   //function evals
379             r8vec_median_estimate(nsamp, meddsc, meddsc_med);   //wrong digits
380
381             //median reliability across the samples for a family
382             medrel = medrel / (pi_base_t) (nsamp);
383
384             //calculate quality value
```

```
385             qality = 0.0;
386             if (medcls_med[0] != 0.0)
387             {
388                 qality = (medact_med[0] + 1.0) * (medest_med[0] + 1.0 - meddsc_med
    [0]) / log(medcls_med[0]);
389             }
390 
391             /*
392             *   fill in DIMENSION SPECIFIC MEDIANS
393             *   (dim-specific medians at the end of each table)
394             */
395             trelib[it] = medrel;
396             tqualt[it] = qality;
397             tactrs[it] = medact_med[1];
398             testrs[it] = medest_med[1];
399             terdsc[it] = meddsc_med[1];
400             tcalsa[it] = medcls_med[1];
401             tcalsb[it] = medcls_med[2];
402             tactrb[it] = medact_med[2];
403             testrb[it] = medest_med[2];
404             terdsb[it] = meddsc_med[2];
405 
406             //SPECIAL CASE: of ndim = 1, the result struct must be filled in here
407             if (ndiml == 1)
408             {
409                 rows[it].rule = rule;
410                 rows[it].fam = genz_name(it + 1);
411                 rows[it].reliability = medrel;
412                 rows[it].int_evals[0] = medcls_med[1];
413                 rows[it].int_evals[1] = medcls_med[2];
414                 rows[it].quality = qality;
415             }
416 
417             /*
418             *   fill in FAMILY-WIDE CONFIDENCE INTERVALS
419             *   (used for final summary of each family)
420             */
421 
422             //fill in arrays for the corresponding dimension and family
423             ersrel[itest - 1 + ndimv * tstlim] = medrel;
424             qualty[itest - 1 + ndimv * tstlim] = qality;
425             ersest[itest - 1 + ndimv * tstlim] = medest_med[1];
426             ersact[itest - 1 + ndimv * tstlim] = medact_med[1];
427             ersdsc[itest - 1 + ndimv * tstlim] = meddsc_med[1];
428             ersesb[itest - 1 + ndimv * tstlim] = medest_med[2];
429             ersacb[itest - 1 + ndimv * tstlim] = medact_med[2];
430             ersdsb[itest - 1 + ndimv * tstlim] = meddsc_med[2];
431             callsa[itest - 1 + ndimv * tstlim] = medcls_med[1];
432             callsb[itest - 1 + ndimv * tstlim] = medcls_med[2];
433 
434             //PRINT A DIM-FAM TABLE ROW
435             rcalsa = (int) medcls_med[1];
436             rcalsb = (int) medcls_med[2];
437             name = genz_name(itest);
438             if (rank == 0)
439             {
440                 printf("%4d  %14s  "PI_F(4, 2) PI_F(5, 2) PI_F(5, 2) PI_F(5, 2
441                        ) PI_F(5, 3) PI_F(4, 2) PI_F(4, 2
442                            ) "  %7d  %7d  "PI_F(6, 3) "  %5d\n", ndim, name,
```

```
443                                 medest_med[1], medest_med[2], medact_med[1],
444                                 medact_med[2], medrel, meddsc_med[1],
445                                 meddsc_med[2], rcalsa, rcalsb, qality, ifails);
446             }
447
448             free(name);
449         }   //END FAMS LOOP
450
451         //calculates the results across all dimensions tested
452         r8vec_median_estimate(tstlim, tactrs, tactrs_med);
453         r8vec_median_estimate(tstlim, trelib, trelib_med);
454         r8vec_median_estimate(tstlim, testrs, testrs_med);
455         r8vec_median_estimate(tstlim, terdsc, terdsc_med);
456         r8vec_median_estimate(tstlim, tactrb, tactrb_med);
457         r8vec_median_estimate(tstlim, testrb, testrb_med);
458         r8vec_median_estimate(tstlim, terdsb, terdsb_med);
459         r8vec_median_estimate(tstlim, tqualt, tqualt_med);
460         r8vec_median_estimate(tstlim, tcalsa, tcalsa_med);
461         r8vec_median_estimate(tstlim, tcalsb, tcalsb_med);
462
463         //PRINT MEDIANS FOR A DIM
464         rcalsa = (int) tcalsa_med[0];
465         rcalsb = (int) tcalsb_med[0];
466         if (rank == 0)
467         {
468             printf("-------------------------------------------------------
469             ------------------------------------------------------\n");
470             printf("%4d   Medians       "PI_F(4, 2) PI_F(5, 2) PI_F(5, 2) PI_F(
471                     5, 2) PI_F(5, 3) PI_F(4, 2) PI_F(4, 2) "%7d  %7d  "PI_F(
472                     6, 3) "\n", ndim, testrs_med[0], testrb_med[0],
473                   testrs_med[0], tactrb_med[0], trelib_med[0], terdsc_med[0],
474                   terdsb_med[0], rcalsa, rcalsb, tqualt_med[0]);
475             printf("\n");
476         }
477
478         free(a);
479         free(alpha);
480         free(b);
481         free(beta);
482     }       //END NDIMS LOOP
483
484     if (1 < ndiml)
485     {
486         if (rank == 0)
487         {
488             printf("-------------------------------------------------------
489             ------------------------------------------------------\n");
490             printf("\n      %s Rule %d Results: \n\n", sbname, rule);
491             printf("-------------------------------------------------------
492             ------------------------------------------------------\n");
493
494             printf("\n\n");
495
496             //print header for the tables to follow
497             printf("        Integrand       Correct digits            Relia-  Wrong
        Integrand Evals  Quality    \n");
498             printf("        Family          Estimated   Actual        bility  Digits
                                    \n\n");
499         }
```

```
500
501         for (it = 0; it < tstlim; it++)
502         {
503             itest = tstfns[it];
504
505             for (j = 0; j < ndiml; j++)
506             {
507                 //pull out familiy specific test-wide results (1 array per family
    containing arrays of results for each dimension tested)
508                 medact[j] = ersact[itest - 1 + j * tstlim];
509                 medest[j] = ersest[itest - 1 + j * tstlim];
510                 meddsc[j] = ersdsc[itest - 1 + j * tstlim];
511                 medacb[j] = ersacb[itest - 1 + j * tstlim];
512                 medesb[j] = ersesb[itest - 1 + j * tstlim];
513                 meddsb[j] = ersdsb[itest - 1 + j * tstlim];
514                 medrll[j] = ersrel[itest - 1 + j * tstlim];
515                 qallty[j] = qualty[itest - 1 + j * tstlim];
516                 medcla[j] = callsa[itest - 1 + j * tstlim];
517                 medclb[j] = callsb[itest - 1 + j * tstlim];
518             }
519
520             //calculate the median values for the family, regardless of dimension
521             r8vec_median_estimate(ndiml, medrll, medrll_med);
522             r8vec_median_estimate(ndiml, medact, medact_med);
523             r8vec_median_estimate(ndiml, medest, medest_med);
524             r8vec_median_estimate(ndiml, meddsc, meddsc_med);
525             r8vec_median_estimate(ndiml, medacb, medacb_med);
526             r8vec_median_estimate(ndiml, medesb, medesb_med);
527             r8vec_median_estimate(ndiml, meddsb, meddsb_med);
528             r8vec_median_estimate(ndiml, qallty, qallty_med);
529             r8vec_median_estimate(ndiml, medcla, medcla_med);
530             r8vec_median_estimate(ndiml, medclb, medclb_med);
531
532             //fill in the struct row to return for summarizing results
533             rows[it].rule = rule;
534             rows[it].fam = genz_name(it + 1);
535             rows[it].reliability = medrll_med[0];
536             rows[it].int_evals[0] = rcalsa;
537             rows[it].int_evals[1] = rcalsb;
538             rows[it].quality = qallty_med[0];
539
540             //PRINT FAMILY-WIDE CONF. INT. MEDIANS
541             rcalsa = (int) medcla_med[0];
542             rcalsb = (int) medclb_med[0];
543             name = genz_name(itest);
544             if (rank == 0)
545             {
546                 printf("      %14s "PI_F(4, 2) PI_F(5, 2) PI_F(5, 2) PI_F(5, 2
547                         ) PI_F(5, 3) PI_F(4, 2) PI_F(4, 2) "%7d  %7d  "PI_F(6, 3
548                             ) "\n", name, medest_med[0], medesb_med[0],
549                                 medact_med[0], medacb_med[0], medrll_med[0],
550                                 meddsc_med[0], meddsb_med[0], rcalsa, rcalsb,
551                                 qallty_med[0]);
552             }
553
554             free(name);
555
556             //fill in family-specific medians
557             tactrs[it] = medact_med[0];
```

```
            testrs[it] = medest_med[0];
            terdsc[it] = meddsc_med[0];
            tactrb[it] = medacb_med[0];
            testrb[it] = medesb_med[0];
            terdsb[it] = meddsb_med[0];
            tcalsa[it] = medcla_med[0];
            tcalsb[it] = medclb_med[0];
            trelib[it] = medrll_med[0];
            tqualt[it] = qallty_med[0];
        }

        //calculate medians for test-wide results, regardless of both family and dimension
        r8vec_median_estimate(tstlim, tactrs, tactrs_med);
        r8vec_median_estimate(tstlim, testrs, testrs_med);
        r8vec_median_estimate(tstlim, terdsc, terdsc_med);
        r8vec_median_estimate(tstlim, tactrb, tactrb_med);
        r8vec_median_estimate(tstlim, testrb, testrb_med);
        r8vec_median_estimate(tstlim, terdsb, terdsb_med);
        r8vec_median_estimate(tstlim, trelib, trelib_med);
        r8vec_median_estimate(tstlim, tqualt, tqualt_med);
        r8vec_median_estimate(tstlim, tcalsa, tcalsa_med);
        r8vec_median_estimate(tstlim, tcalsb, tcalsb_med);

        //PRINT GLOBAL MEDIANS
        rcalsa = (int) tcalsa_med[0];
        rcalsb = (int) tcalsb_med[0];
        if (rank == 0)
        {
            printf("------------------------------------------------------------
            ------------------------------------------------------------\n");
            printf("       Global medians "PI_F(4, 2) PI_F(5, 2) PI_F(5, 2
                   ) PI_F(5, 2) PI_F(5, 3) PI_F(4, 2) PI_F(4, 2
                       ) "%7d  %7d  "PI_F(6, 3) "\n", testrs_med[0],
                           testrb_med[0], tactrs_med[0], tactrb_med[0],
                           trelib_med[0], terdsc_med[0], terdsb_med[0], rcalsa,
                           rcalsb, tqualt_med[0]);
            printf("\n");
        }
    }

    free(medact);
    free(medcls);
    free(meddsc);
    free(medest);
    free(medrll);
    free(qallty);

    return rows;
}

void test(int rule, int ndims, pi_base_t a[], pi_base_t b[], int maxpts,
          pi_ifcn_t integrand, pi_base_t rel_err_tol, pi_base_t *rel_err,
          pi_base_t *f_int_est, int *eval_lim_flag, int *eval_count)
/**************************************************************************
    test()
    -used by multst to evaluate an integrand numerically via ParInt's API
    -performs a single integration, called 20 times
     per ndim + integrand family combination.
```

```
    INPUT:
        ndims ____________ # of dimensions to integrate over
        a[], b[] _________ bounds of integration for each dimension
                           (need to be of length ndims)
        maxpts ___________ # of integrand evaluations allowed (400,000)
        integrand ________ the function (integrand) to be integrated numerically
                        (see genz_functions() for paramater details)
        rel_err_tol ______ the relative error tolerance requested (1.0E-06)
    OUTPUT:
        rel_err __________ estimation of the relative error
                           (estimates how accuracte the result is likely to be)
        f_int_est ________ result of the numerical integration (this is approximate,
     not necessarily exact)
        eval_lim_flag ____ set to 0 if maxpts # of evals is not passed, set to 1 if
    it is passed
        eval_count _______ allows the # of integrand evaluations to be recorded and
    returned to mltst()
                           (normally this info is printed with pi_print_results and
    a *pi_status_t)
*************************************************************************/
{
    int             i;
    pi_status_t     status;
    pi_hregion_t    *hrgn_p;
    pi_base_t       eps_a = rel_err_tol;    //absolute error tolerance
    pi_base_t       eps_r = rel_err_tol;    //relative error tolerance  --must be
    greater than the machine percision of the pi_base_t being used
    hrgn_p = pi_allocate_hregion(ndims);    //allocate region to integrate over
    for (i = 0; i < ndims; i++)
    {
        hrgn_p->a[i] = a[i];
        hrgn_p->b[i] = b[i];
    }

    pi_integrate(integrand, 1, rule, maxpts, eps_a, eps_r, hrgn_p, PI_RGN_HRECT,
                 f_int_est, rel_err, &status);

    *eval_lim_flag = status.fcn_limit_flag; //if the eval count was surpassed, set
    to 1, 0 otherwise
    *eval_count = status.fcn_eval_count;

    pi_free_hregion(hrgn_p);                //free memory allocated for integration
    region
}

void setup(int nsamp, int tstlim, pi_base_t *concof, int *nconf,
           pi_base_t *small, int *idfclt, int *tstfns, pi_base_t *difclt,
           pi_base_t *expons, pi_base_t *expnts)
/*************************************************************************
    setup()
        initialize and compute confidence coefficent,
        exponents, and difficulty levels for the integrand families
*************************************************************************/
{
    //Initialize and compute confidence coefficient.
    //C > 9.5 for S > 5 (C being confidence coefficent, S being sample size)
    int i;
    *concof = 0.0;
    *nconf = i4_max(1, (2 * nsamp) / 5 - 2);
```

```
668
669     for (i = 1; i <= *nconf; i++)
670     {
671         *concof = 1.0 + (pi_base_t) (nsamp - (*nconf) + i) * (*concof) / (pi_base_t)
        ((*nconf) - i + 1);
672     }
673
674     *concof = 1.0 - (*concof) / (pi_base_t) (i4_power(2, nsamp - 1));
675
676     *small = r8_epsilon();
677
678     for (i = 0; i < tstlim; i++)
679     {
680         idfclt[i] = (int) difclt[tstfns[i] - 1];
681     }
682
683     for (i = 0; i < tstlim; i++)
684     {
685         expons[i] = expnts[tstfns[i] - 1];
686     }
687 }
688
689 void print_init(int rule, int ndims, int *dims, int nsamp, int tstlim,
690                 int maxpts, int *idfclt, pi_base_t *expons, int *digits,
691                 char *sbname, pi_base_t concof, pi_base_t rel_tol)
692 /**********************************************************************
693     print_init()
694         -displays initial test information, including:
695         # of samples
696         difficulty levels w/ corresponding exponents
697         requested # of digits
698         max # of function evals
699         confidence coefficent of the tests
700         name of the test method
701 **********************************************************************/
702 {
703     int j;
704     printf("\n\n\n++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
705     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++");
706     printf("\n\n     Test Paramaters for %s Rule %d: \n\n", sbname, rule);
707     printf("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
708     ++++++++++++++++++++++++++++++++++++++++++++++++++\n");
709
710     printf("\n   Showing test results with %d samples per test.\n", nsamp);
711
712     printf("\n   Testing the following dimensions:\n\n");
713     for (j = 0; j < ndims; j++)
714     {
715         printf("%5d", dims[j]);
716     }
717
718     printf("\n\n   Difficulty levels");
719     for (j = 0; j < tstlim; j++)
720     {
721         printf("%6d", idfclt[j]);
722     }
723
724     printf("\n           Exponents");
725     for (j = 0; j < tstlim; j++)
```

```
    {
        printf(PI_F(3, 8), expons[j]);
    }

    printf("\n");
    *digits = (int) (-log10(rel_tol));
    printf("\n   Requested digits = %d           Maximum values = %d\n\n",
           *digits, maxpts);
    printf("   Produces variable results with confidence = "PI_F(6, 0) "\n",
           concof);
    printf("\n------------------------------------------------------------
    ------------------------------------------\n\n");

    //print header for the tables to follow
    printf(" N-Dims Integrand       Correct digits            Relia-  Wrong
    Integrand Evals  Quality    Total\n");
    printf("        Family          Estimated   Actual        bility  Digits
                          Fails\n\n");
}

void generate_params(int ndim, int itest, pi_base_t *a, pi_base_t *b,
                     pi_base_t *total, pi_base_t *dfact, pi_base_t dfclt,
                     pi_base_t exn)
/*****************************************************************************
    generate_params()
        -called for each {ndim, rule} combination being tested, initializes the
    paramaters
        for the nsamp tests (default 20)
        -initializes:
        alpha and beta (free paramater vectors)
        a and b (region limit vectors)

        -also normalizes some of alpha, beta, a and b, for specific integrand
    families.
        this is done using the difficulty levles and exponents
*****************************************************************************/
{
    /*
      Choose the integrand function parameters at random.
      Values will be between [0,1]
      ALPHA - the affective paramaters (don't affect all families,
      affect difficulty of integration, so gets normed)
      BETA - the unaffective paramaters, they should remain random and unmodified
    */
    int seed = 123456;                      //same seed used for all tests for
    reproducability
    int n,
        j;

    for (n = 0; n < ndim; n++)
    {
        alpha[n] = genz_random(&seed);
        beta[n] = genz_random(&seed);
    }

    /*
      Modify ALPHA to account for difficulty parameter.
    */
    *total = r8vec_sum(ndim, alpha);
```

```
780     *dfact = *total * pow(ndim, exn) / dfclt;
781     for (j = 0; j < ndim; j++)
782     {
783         alpha[j] = alpha[j] / *dfact;
784     }
785 
786     /*
787       For families 1 and 3, we modify the value of B.
788       (this has seemingly no effect on the integration, it is reducing the interval
    of integration
789       as well as the difficulty, which is essentially just an equivilent
    transformation)
790     */
791     if (itest == 1 || itest == 3)
792     {
793         for (j = 0; j < ndim; j++)
794         {
795             b[j] = alpha[j];
796         }
797     }
798 
799     /*
800       For test 6, we modify the value of BETA.
801       (this is because the values of beta beyond X2 are irrelevent, we only care
    about the first 2
802       so set these others to known quantities)
803     */
804     if (itest == 6)
805     {
806         for (n = 2; n < ndim; n++)
807         {
808             beta[n] = 1.0;
809         }
810     }
811 }
812 
813 void write_result(struct Result_Row *rows, FILE *fp, int n_fams)
814 /*************************************************************************
815 *************************************************************************/
816 {
817     int i;
818     for (i = 0; i < n_fams; i++)
819     {
820         fprintf(fp, "%d  %s  "PI_F(1, 20) "%d  %d  "PI_F(2, 20) "\n",
821                 rows[i].rule, rows[i].fam, rows[i].reliability,
822                 rows[i].int_evals[0], rows[i].int_evals[1], rows[i].quality);
823     }
824 }
```