Dalsoft's Random Testing

Reference Manual

version 3.0
for the Linux/Windows operating systems

www.products.dalsoft.com/drt.html

General

drt - Dalsoft's Random Testing package provides a framework for automated software testing (**fuzzing);** it permits the repeated execution of user-provided algorithms allowing to control the execution by specifying

- means to establish input data
- termination condition and/or execution duration (time and/or number of cases)
- reporting execution statistics

It provides a C++ class which may be used to establish code execution frame.

drt may be particularly useful in number of ways:

- to run the code in controlled manner
- to recreated the case(s) were a problem was detected for further investigation (e.g. debugging)
- to study the implemented algorithm and to obtain statistics about it

drt also provides a standalone random numbers generator that allows to control random numbers generation and to generate random numbers of a various kind. This standalone random numbers generator may be used in C and C++ programs.

Technical specifications

Current version:	3.0
Source code in:	C++,C
Operating system supported:	Linux, Windows
Documentation:	manual
Support:	on-line

Installation

The release package includes:

drt.h – include file for Linux/Windows

linux_libdrt.a – library for Linux

windows_libdrt.lib – library for Windows.

To install ***drt***, copy the provided library (e.g. ***linux_libdrt.a***) and the include file (***drt.h***) into the appropriate working directories: on Linux it may be */usr/local/lib64* for ***libdrt.a*** and */usr/local/include* for ***drt.h*** renaming the library as ***libdrt.a***; e.g. on Linux do:

```
sudo cp linux_libdrt.a /usr/local/lib64/libdrt.a
```

Compiling and linking with the drt library

This section describes how to use the ***drt*** with your programs.

To use ***drt*** in C++ programs, you must **#include** the file **drt.h** in every source file that calls ***drt*** functions, accesses ***drt*** global variables, or uses constants defined by ***drt***.

The procedures for linking ***drt*** with the rest of your program vary according to the compiler and method you are using. For example

```
g++ mandyprogram.cpp -ldrt
```

It was observed that on Linux, in certain cases, it is necessary to specify -no-pie option while linking object files with the ***drt*** library.

Random numbers generation

drt is based on the ability to generate random data of different kind and currently provides two types of random number generation engines

1. ***stc***: based on the routine ***rand*** and related to it from the standard C library.
2. ***mt19937***: a variant of the twisted generalized feedback shift-register algorithm that is known as the "Mersenne Twister" generator.

Random number generation of different data types is based on these random number generation engines.
Although there are no convincing guarantees about the quality of random numbers returned, it should be good enough for a casual use.

drt supports two kind of random generation: ***implicit*** and ***explicit***. Following there is a detailed explanation about these functionality, here we just provide a brief overview.

- ***implicit***
  drt keeps only one instance of random generator, The internals are hidden from the user.

This kind of random number generation is not thread safe and may not be used in multi-threaded (parallel) environment.

- ***explicit***
  drt allows to create and use many instances of random generator. The user is responsible for creation, maintenance and destruction of these instances of random generator. The internals are mostly hidden from the user. This kind of random number generation is thread safe and may be used in multi-threaded (parallel) environment.

Note that ***drt*** itself uses only ***implicit*** random generation. The standalone random data generator, offered by ***drt***, allows for both - ***implicit*** and ***explicit*** - kinds of random generators to be used.

Evaluation of the random number generation engines

The following are some of the results of evaluation of these random number generation engines - we generated one million samples using these engines and analyzed the results as described here. The ***stc*** random number generation engine was based on the GNU libc version 2.33.

Data analysis:

stc

```
Entropy = 7.954457 bits per byte.

Optimum compression would reduce the size
of this 400000000 byte file by 0 percent.

Chi square distribution for 400000000 samples is 24987700.24, and
randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 111.5039 (127.5 = random).
Monte Carlo value for Pi is 3.485869115 (error 10.96 percent).
Serial correlation coefficient is -0.049173 (totally uncorrelated = 0.0).
```

mt19937

```
Entropy = 7.999999 bits per byte.

Optimum compression would reduce the size
of this 400000000 byte file by 0 percent.

Chi square distribution for 400000000 samples is 290.56, and randomly
would exceed this value 6.23 percent of the times.

Arithmetic mean value of data bytes is 127.5022 (127.5 = random).
Monte Carlo value for Pi is 3.141552871 (error 0.00 percent).
Serial correlation coefficient is -0.000037 (totally uncorrelated = 0.0).
```

It shall be pointed out that ***stc*** generates 31 bits random numbers while ***mt19937*** generates 32 bits random numbers - this favors results of data analysis of ***mt19937*** over ***stc***.

Using drt

Later in this document we provide the detailed example of ***drt*** use. Here we outline the general guidance for this.

drt provides C++ class **CDRT**. One of the ways to use ***drt*** is for user to define it own class derived from **CDRT**:

```
class MyTest : public CDRT
 {
...
```

and then

```
int main( int ac, char **av )
 {
  MyTest rt;
  int stat;

  stat = rt.m_GetOptions( ac, av );  ← defined in CDRT
  if ( stat == DRT_OPTION_EXIT_OK )
   {
    exit( 0 );
   }
  else if ( stat == DRT_OPTION_EXIT_ERROR )
   {
    exit( 1 );
   }

   // if ( stat == DRT_OPTION_OK )
  rt.m_RunTest();  ← defined in CDRT

  exit( 0 );

 }  /*  main  */
```

Programming with drt

This section describes the members of the class **CDRT** provided by ***drt***.

Working Environment

drt allows to establish means to control the execution of your algorithm as following. The default values are provided, the means to overwrite these defaults is possible by specifying input at the invocation of the program and/or calling provided by ***drt*** routine ***m_GetOptions***. Note that ***drt***

utilizes ***implicit*** kind of random generation.

controlling duration of the executions

The program run duration is controlled by the following parameters; program execution is halted by the first fulfilled parameter.

-t #   - time in seconds to run test for, 0 - unlimited; test runs for 10 seconds if this parameter is not specified`
-c #   - number of cases to execute, unlimited if this parameter is not specified
-sfe   - request to stop on the first error

Note that for time calculation ***drt*** uses the total amount of time spent executing in user mode or, if not available, the processor time consumed by the program. For multi-thread programs, this may differ from the time spent by the program execution.

establishing initial environment

If using random number generator provided by ***drt***, in order to be able to recreate the running data use the parameter -nsi # - generate new seed for random number generator using the specified argument as a seed. Specifying -nsi # with the same parameter causes the same random numbers to be generated thus allowing to repeat test execution on the same data.
If using random number generator provided by ***drt***, in order to establish different running data for different runs, use the parameter -ns - generate new seed for random number generator using random seed.

In order to establish (recreate) execution environment the parameters -s # and -srn # may be used. Note that the parameter -s # causes the function **m_GenRandData** to be called the specified number of times - the accuracy of such an approach depends on the implementation of this function. The more reliable way to recreate execution environment is to use the parameter -srn # that causes the random number generator of the established engine to be called the specified number of times; use **drt_rand_getRandNumbersGenerated()** to obtain the number of calls that are performed and is desirable to skip on the next invocation. Also you may use ***m_GetRandNumbersGeneratedBeforeCrntTest*** and print it in the test report (see ***m_DumpReport()***) and later skip that number of calls to arrive at the desirable data.

Member functions

General Control

m_GetOptions

int m_GetOptions(int ac, char **av)

m_GetOptions assumes it arguments **ac** and **av** to be C-style arguments similar to those **main** is called to begin execution

1. **ac** is number of arguments in **av** to be processed

2. **av** is a pointer to an array of character strings that contain the arguments, one per string

The following is a list of the parameters and descriptions of their functionality:

-t # - time in seconds to run test for, 0 - unlimited; test runs for 10 seconds if this parameter is not specified
-c # - number of cases to execute; unlimited if this parameter is not specified
-s # - number of cases to skip; 0 if this parameter is not specified
-srn # - number of calls to the random number generator to skip; 0 if this parameter is not specified
-ns - generate new seed for random number generator using random seed
-nsi # - generate new seed for random number generator using the specified argument as a seed
-sfe - request to stop on the first error
-rng_stc - establish ***stc*** as the engine for random number generation (explained later)
-rng_mt19937 - establish ***mt19937*** as the engine for random number generation (explained later)
-param0 #...-param9 # - accept a numeric parameter that may be used by the implemented code
-h - prints short help message and exits
-help - prints short help message and exits

Return values:

DRT_OPTION_OK - all parameters were properly specified
DRT_OPTION_EXIT_OK - all parameters were properly specified and terminating parameter (e.g. -help) was identified
DRT_OPTION_EXIT_ERROR - parameters specified couldn't be parsed

m_GetParam

long m_GetParam(unsigned int indx) const;

Return the value of the command line parameter -param<indx> #, e.g **m_GetParam(2)** returns the value of the command line parameter -param2; if parameter is not specified or wrong index - **0** is returned.

Running and monitoring test executions

m_GenRandData

virtual void m_GenRandData() = 0;

Routine to initialize data for execution/verification of a test. Initializes underlying data records and **MUST BE PROVIDED BY THE USER.**

m_ShallTerminateRunTest

virtual int m_ShallTerminateRunTest();

Return not a zero value if test execution shall be terminated, zero value otherwise.

The standard (provided by ***drt***) implementation of this routine trigers termination of the test execution
if interrupt (e.g. Cntrl-C) was detected or of the option -sfe (request to stop on the first error) was established and an error was detected (e.g. failure of the current test).

m_TestRandData

virtual int m_TestRandData() = 0;

Routine that executes the test for the currently established data and verifies the result(s) of execution. **MUST BE PROVIDED BY THE USER.**

Return values:

DRT_ATTEMPTED_RUN_OK - the current test was successfully executed and the results of execution were verified without any error being detected
DRT_ATTEMPTED_RUN_FAILED - the current test was successfully executed and the results of execution were verified with error(s) being detected
DRT_ATTEMPTED_NOT_RUN - the execution of the current test was not attempted or shall be ignored

m_RunTest

virtual void m_RunTest();

Routine that initializes data for execution/verification of a test, executes the test for the currently established data and verifies the result(s) of execution. May be provided by the user.

m_GetCrntTestNumber

unsigned long m_GetCrntTestNumber() const;

Return the test number of the test for which the data being established or which is currently executed.
The test are numbered beginning from **0** (test **1** having current test number being **0**).
The standard (provided by ***drt***) implementation of the routine **m_RunTest()** initializes and updates test numbers. If the routine **m_RunTest()** is overwritten, the return value of this function is undefined.

m_GetRandNumbersGeneratedBeforeCrntTest

unsigned long m_GetRandNumbersGeneratedBeforeCrntTest() const;

Return the number of calls made to the random number generator before the current test. May be useful to recreate execution environment using -srn # parameter.

m_DumpReport

void m_DumpReport();

Prints the general statistics accumulated by ***drt*** during the execution.
The printing information includes:

Total - total number of cases that were processed: number of cases that were skipped plus number of cases that were attempted to be executed
Attempted - number of cases that were attempted to be executed
Done - number of cases that were actually executed
Execution time - time in seconds that too to execute test cases; doesn't include time that took to skip unwanted cases
error count - number of times **m_TestRandData** returned **DRT_ATTEMPTED_RUN_FAILED**
Random number generator seed used - invoking the package with the nsi parameter being equal to the printed value allows the same data to be generated

For example, the following is output generated by the **m_DumpReport** routine with ***drt*** being invoked in the following way:

```
-s 120 -t 10
```

(skip **120** cases and run for **10** seconds)

```
Total 136, Attempted 16, Done 16 cases for 10.1676 seconds error count = 11.
Random number generator seed used 1588429226.
```

Standalone random data generator

drt may be used as a general purpose random data generator for C and C++ code.

drt supports two kind of random generation: ***implicit*** and ***explicit***.

implicit random generation

In ***implicit*** random generator ***drt*** keeps only one instance of random generator. The internals are hidden from the user. The name of every implicit random generation routine begins with ***drt_rand_***, e.g ***drt_rand_int***.

This kind of random number generation is not thread safe and may not be used in multi-threaded (parallel) environment.

Establishing random number generation engine

The following are the routines that allow to establish random number generation engine to be used in a standalone random numbers generator. The default random number generator provided by ***drt*** is ***stc***.

drt_rand_establishRNG_stc

void drt_rand_establishRNG_stc();

Establishes the random number generation engine to be based on the routine ***rand*** (and related to it) from the standard C library.

drt_rand_establishRNG_mt19937

void drt_rand_establishRNG_mt19937();

Establishes the random number generation engine to be based on the a variant of the twisted generalized feedback shift-register algorithm, and is known as the “Mersenne Twister” generator. Based on the code from here (which also contains disclaimer and the copyright message).

drt_rand_establishRNG

void drt_rand_establishRNG(int arg);

Establishes the random number generation engine as following:
if **arg** is equal to **RNG_mt19937** then ***mt19937*** random number generation engine is established to be used, otherwise ***stc*** random number generation engine is established to be used.

drt_rand_determineEstablishedRNG

int drt_rand_determineEstablishedRNG()

Returns the value that indicates which random number generation engine is used:
  RNG_stc if ***stc*** random number generation engine is used
  RNG_mt19937 if ***mt19937*** random number generation engine is used

Functions for random data generation

drt_rand_init

unsigned int drt_rand_init(unsigned int val);

Generate new seed for random number generator using the specified argument **val** as a seed . Returns the seed used. Routines provided by ***drt*** guarantee that using the same argument to this function will result in the same sequence of random numbers generated.

drt_rand_determineUsedSeed

unsigned int drt_rand_determineUsedSeed();

Returns the value that was used as a seed for random number generation engine. If no seed was explicitly used then **0** is returned.

drt_rand_getRandNumbersGenerated

unsigned long drt_rand_getRandNumbersGenerated();

Returns the number of time the random number generator of the established engine was called.

drt_rand_int

int drt*_rand_int*();

Return random number of the type **int**.

drt_rand_RAW

int drt*_rand_RAW*();

Return random number of the type **int** exactly as it is generated by the underlying random generation engine - no processing is attempted.

drt_rand_long

long drt_rand*_long*();

Return random number of the type **long**.

drt_rand_bool

int m_rand_bool();

Return random number of the type **int** that may be equal to **0** or **1**.

drt_rand_float

float drt_rand_float();

Return random number of the type **float**.

drt_rand_double

***double* drt_rand_double();**

Return random number of the type **double**.

drt_rand_intRange

int drt_rand_intRange(int rMin,* int *rMax);

Return random number of the type **int** in the range between **rMin** and **rMax**. If **rMin**, **rMax** specify not a valid range (e.g. **rMin** > **rMax**) - return a random number of the type **int**.

drt_rand_uintRange

unsigned int drt_rand_uintRange(* unsigned *int rMin, unsigned int rMax);

Return random number of the type **unsigned int** in the range between **rMin** and **rMax**. If **rMin**, **rMax** specify not a valid range (e.g. **rMin** > **rMax**) - return a random number of the type **unsigned int**.

drt_rand_ulongRange

unsigned int drt_rand_ulongRange(* unsigned *long rMin, unsigned long rMax);

Return random number of the type **unsigned long** in the range between **rMin** and **rMax**. If **rMin**, **rMax** specify not a valid range (e.g. **rMin** > **rMax**) - return a random number of the type **unsigned long**.

drt_rand_intInitElements

void drt_rand_intInitElements(init *x, unsigned long firstIndx, unsigned long cnt, int rMin, int rMax);

Starting from the element at the index **firstIndx** sets **cnt** elements of the array **x** of the integer members to a random integer values in the range between **rMin** and **rMax**.

drt_rand_doubleRange

double drt_rand_doubleRange(double rMin, double rMax);

Return random number of the type **double** in the range between **rMin** and **rMax**. If **rMin**, **rMax** specify not a valid range (e.g. **rMin** > **rMax**) - return a random number of the type **double**.

drt_rand_doubleInitElements

void drt_rand_doubleInitElements(double *x, unsigned long firstIndx, unsigned long cnt, double rMin, double rMax);

Starting from the element at the index **firstIndx** sets **cnt** elements of the array **x** of the double precision members to a random double precision values in the range between **rMin** and **rMax**.

explicit random generation

In ***explicit*** random generator ***drt*** allows to create and use many instances of random generator. The user is responsible for creation, maintenance and destruction of these instances of random generator. The internals are mostly hidden from the user.

drt provides special data type ***drt_rand*** which is used as a handler for a random generator. Each ***explicit*** random generator shall be allocated by user calling ***drt_rand_alloc_handler*** which returns the value of the type ***drt_rand*** called 'handler' of the created random generator. This value shall be used as the (first) parameter to most ***explicit*** random generator routine. We will refer to an ***explicit*** random generator by referring to it handler of the type ***drt_rand***.

The name of every ***explicit*** random generation routine begins with ***drt_rand__***, and, in the most cases, the first argument is a variable of the type ***drt_rand*** (the handler) e.g ***drt__rand_int(drtHandler)***. Calling ***explicit*** random generation routine with an invalid handler (e.g. **NULL**) defaults to the corresponding ***implicit*** random generation routine, e.g. instead of ***drt__rand_int(* NULL *) drt_rand_int()*** is executed.

The ***explicit*** random number generation is thread safe and may be used in multi-threaded (parallel) environment.

Establishing random number generation engine

The following are the routines that allow to create and establish ***explicit*** random number generation engine to be used in a standalone random numbers generator. The only type of ***explicit*** random number generator currently provided by ***drt*** is ***mt19937***.

drt_rand__alloc_handler

drt_rand *drt_rand__alloc_handler(int arg);

Establishes the ***explicit*** random number generation engine and returns it handler.
arg specifies the type random generation engine to be created and currently must be equal to **RNG_mt19937**.
The ***explicit*** random number generation engine established by this routine sets it seed to zero (which can be later changed by calling ***drt_rand__init***).
If bad **arg** is specified or there is not sufficient memory to create the generator - **NULL** is returned.

Note that this is the only the ***explicit*** random number generation routine that doesn't require the first argument to be a variable of the type ***drt_rand*** (the handler) .

drt_rand__establishRNG

void drt_rand__establishRNG(drt_rand *rngHandler, int arg);

Establishes the random number generation engine for the handler **rngHandler** as following:
if **rngHandler** is **NULL** then the implicit random generator is affected as following:

> if **arg** is equal to **RNG_mt19937** then ***mt19937*** random number generation engine is established to be used, otherwise ***stc*** random number generation engine is established to be used.

If **rngHandler** is a valid handler returned by ***drt_rand__alloc_handler***, then no action is taken (currently).

Functions for random data generation

drt_rand__init

unsigned int drt_rand__init(drt_rand *rngHandler, unsigned int val);

Generate new seed for random number generator **rngHandler** using the specified argument **val** as a seed .
Returns the seed used. Routines provided by ***drt*** guarantee that using the same argument to this function will result in the same sequence of random numbers generated.

drt_rand__determineUsedSeed

unsigned int drt_rand__determineUsedSeed(drt_rand *rngHandler);

Returns the value that was used as a seed for random number generator **rngHandler**.

drt_rand__getRandNumbersGenerated

unsigned long drt_rand__getRandNumbersGenerated(drt_rand *rngHandler);

Returns the number of times the random number generator **rngHandler** was called.

drt_rand__int

int drt_rand__int(drt_rand *rngHandler);

Return random number of the type **int** generated by the random number generator **rngHandler**.

drt_rand__RAW

int drt_rand__RAW(drt_rand *rngHandler);

Return random number of the type **int** exactly as it is generated by the underlying random generation engine of **rngHandler** - no processing is attempted.

drt_rand__long

long drt_rand__long(drt_rand *rngHandler);

Return random number of the type **long** generated by the random number generator **rngHandler**.

drt_rand__bool

int m_rand__bool(*drt_rand *rngHandler*);

Return random number of the type **int** that may be equal to **0** or **1** generated by the random number generator **rngHandler**.

drt_rand__float

float drt_rand__float(*drt_rand *rngHandler*);

Return random number of the type **float** generated by the random number generator **rngHandler**.

drt_rand__double

***double* drt_rand__double(*drt_rand *rngHandler*);**

Return random number of the type **double** generated by the random number generator **rngHandler**.

drt_rand__intRange

int drt_rand__intRange(drt_rand *rngHandler*, *int rMin*, int *rMax);

Return random number of the type **int** generated by the random number generator **rngHandler** in the range between **rMin** and **rMax**. If **rMin**, **rMax** specify not a valid range (e.g. **rMin** > **rMax**) - return a random number of the type **int** generated by the random number generator **rngHandler**.

drt_rand__uintRange

unsigned int drt_rand__uintRange(drt_rand *rngHandler*, unsigned *int rMin*, *unsigned int rMax);

Return random number of the type **unsigned int** generated by the random number generator **rngHandler** in the range between **rMin** and **rMax**. If **rMin**, **rMax** specify not a valid range (e.g. **rMin** > **rMax**) - return a random number of the type **unsigned int** generated by the random number generator **rngHandler**.

drt_rand__ulongRange

unsigned int drt_rand__ulongRange(drt_rand *rngHandler*, unsigned *long rMin*, *unsigned long rMax);

Return random number of the type **unsigned long** generated by the random number generator **rngHandler** in the range between **rMin** and **rMax**. If **rMin**, **rMax** specify not a valid range (e.g. **rMin** > **rMax**) - return a random number of the type **unsigned long** generated by the random number generator **rngHandler**.

drt_rand__intInitElements

void drt_rand__intInitElements(*drt_rand *rngHandler*, init *x, unsigned long firstIndx, unsigned long cnt, int rMin, int rMax);

Starting from the element at the index **firstIndx** sets **cnt** elements of the array **x** of the integer members to a random integer values in the range between **rMin** and **rMax**.

drt_rand__doubleRange

double drt_rand__doubleRange(*drt_rand *rngHandler*, double rMin, double rMax);

Return random number of the type **double** generated by the random number generator **rngHandler** in the range between **rMin** and **rMax**. If **rMin**, **rMax** specify not a valid range (e.g. **rMin** > **rMax**) - return a random number of the type **double** generated by the random number generator **rngHandler**.

drt_rand__doubleInitElements

void drt_rand__doubleInitElements(*drt_rand *rngHandler*, double *x, unsigned long firstIndx, unsigned long cnt, double rMin, double rMax);

Starting from the element at the index **firstIndx** sets **cnt** elements of the array **x** of the double precision members to a random double precision values generated by the random number generator **rngHandler** in the range between **rMin** and **rMax**.

Examples

testing *dco* parallelization of a stencil

The following shows how ***drt*** was used to perform testing and monitoring of the parallel code created by ***dco*** for the serial stencil for double precision values:

```
for ( i = 1; i < DIM; i++ )
 {
  x[i] = x[i] - a[i]/x[i-1];
 }
```

Download the C++ source code for this example from here and the executable for LINUX OS from here.

Description of the problem

dco is a software optimization package created by **Dalsoft** that, among many other functions, performs automatic parallelization of a serial code - see this for more information.
When the parallel code for the above stencil was created it was necessary to verify that created code was correct. That was done in the early stages of testing; after that we decided to find out how accurate the parallel code is - that is what we are going to show here.

Three execution results were created:

- ***dco result*** - result of the execution by the parallel code created by ***dco*** for the above stencil
- ***exact result*** - result of the execution of the above code using double precision values
- ***precise result*** - result of the execution of the above code using Dalsoft High Precision (***dhp***) package - see this for more information.

dco result is generated by the parallel code and is fast. ***exact result*** is what the standard implementation of the above algorithm generates. Due to the inexact nature of the double precision floating point execution, it is not clear how accurate these results are. We assume that using high precision data allows to generate more accurate - ***precise result***. The following use of ***drt*** attempts to compare these results.

Implementation

The implementation requires

- to create the new class (we will call it **STENCILTest**) derived from **CDRT**
- to establish data records to be used in test and implement class member **m_GenRandData()** to initialize this data
- to implement class member **m_TestRandData()** to perform results generation and collect necessary data about results accuracy

- to implement class member **m_DumpReport()** to print the collected information
- to implement function **main** to execute code testing

The file **stenciltest.cpp** that contains implemented code may found here.

new class STENCILTest

```
class STENCILTest : public CDRT
 {

   // Construction
   public:
    STENCILTest();
    ~STENCILTest(){};

   public:
    virtual void m_GenRandData();
    virtual int m_TestRandData();
    virtual void m_DumpReport();

   private:
maximum relative difference between 'dco result'
and 'exact result' that exceeds 'm_dRelativeErrorThreshold'
    double   m_dMaxDiff;
maximum relative difference between 'dco result'
and 'exact result'
    double m_dTotalMaxDiff;

test number were 'm_dMAxDiff' was detected
    unsigned long m_nTestNumberForMaxDiff;
index inside stencil loop were 'm_dMAxDiff' was detected
    unsigned int m_nIndxInternalForMaxDiff;
'dco result' for the test 'm_nTestNumberForMaxDiff at the index
'm_nIndxInternalForMaxDiff'
    double m_d_dco_result;
'exact result' for the test 'm_nTestNumberForMaxDiff at the index
'm_nIndxInternalForMaxDiff'
    double m_d_exact_result;
'precise result' for the test 'm_nTestNumberForMaxDiff at the index
'm_nIndxInternalForMaxDiff'
    dhpreal m_dhp_precise_result;
maximum relative difference between 'dco result' and 'precise result'
            m_dhpDiffdcoMax,
maximum relative difference between 'eact result' and 'precise result'
            m_dhpDiffexactMax;
number of cases were 'dco result' was closer that 'exact result' to
'precise result'
   unsigned long m_n_dcoBetter;
number of cases were 'exact result' was closer that 'dco result' to
'precise result'
   unsigned long m_n_exactBetter;
number of cases were 'exact result' and 'dco result' equaly close to
'precise result'
   unsigned long m_n_dco_exactSame;

error will be reported if if relative difference between 'exact result' and
'dco result' exceeds this value
   double m_dRelativeErrorThreshold;

 };
```

and the constructor that initializes the fields of the class

```
STENCILTest::STENCILTest()
 {

 m_dMaxDiff = 0.;
 m_dTotalMaxDiff = 0.;

 m_dhpDiffdcoMax = 0.;
 m_dhpDiffexactMax = 0.;
 m_n_dcoBetter = 0;
 m_n_exactBetter = 0;
 m_n_dco_exactSame = 0;

 m_dRelativeErrorThreshold = 10e-16;

 }  /*   STENCILTest::STENCILTest   */
```

establish data records to be used in test and implement class member m_GenRandData() to initialize this data

Define the following records to be used:

```
#define DIM      100000

double a[DIM],   storage for stencils coefficients
       x[DIM],   storage for 'dco result'
       x1[DIM];  storage for 'exact result'
dhpreal aDHP[DIM], xDHP[DIM];storage for calculation of 'precise
result',
```

dhpreal being defined by the Dalsoft High Precision (dhp) package

and implement **m_GenRandData** to randomly set the above records

```
void STENCILTest::m_GenRandData()
 {
 unsigned int i;
```

set DIM elements of the arrays 'x' and 'a' to a random double precision values in the range between 0.1 and 0.2

```
 drt_rand_doubleInitElements( x, 0, DIM, .1, .2 );
 drt_rand_doubleInitElements( a, 0, DIM, .1, .2 );
```

set elements of the array 'x1' and 'xDHP' to those of 'x' and set elements of the array 'aDHP' to those of 'a'

```
 for ( i = 0; i < DIM; i++ )
  {
  x1[i] = x[i];
  xDHP[i] = x[i];

  aDHP[i] = a[i];
  }

 }  /*   STENCILTest::m_GenRandData   */
```

implement class member m_TestRandData() to perform results generation and collect necessary data about results accuracy

```
int STENCILTest::m_TestRandData()
 {
 unsigned int i;
 int ret;

 ret = DRT_ATTEMPTED_RUN_OK;
```

generate 'dco result' in the array 'x' - prepare for parallel code generation; the code will be processed by the Dalsoft's auto-parallelizer (dco); dco will be directed to only process (create parallel code for) the section between ',dco_start' and '.dco_end'

```
asm( "#.dco_start" );
 for ( i = 1; i < DIM; i++ )
  {
   x[i] = x[i] - a[i] / x[i-1];
  }
 asm("#.dco_end");

generate 'exact result' in the array 'x1'
  for (i = 1; i < DIM; i++)
  {
   x1[i] = x1[i] - a[i] / x1[i-1];
  }

generate 'precise result' in the array 'xDHP'
  for (i = 1; i < DIM; i++)
  {
   xDHP[i] = xDHP[i] - aDHP[i] / xDHP[i-1];
  }

  dhpreal xdiffDHP, x1diffDHP;
  double d;

  for (i = 0; i < DIM; i++)
  {
calculate relative difference between 'dco result' and 'precise result'
   xdiffDHP = CalcRelativeDiff(x[i], xDHP[i]);
calculate relative difference between 'exact result' and 'precise result'
   x1diffDHP = CalcRelativeDiff(x1[i], xDHP[i]);

   if (xdiffDHP < x1diffDHP)
    {
relative difference between 'dco result' and 'precise result' is smaller
that relative difference between 'exact result' and 'precise result' thus
'dco result' is "better"
    m_n_dcoBetter++;
    }
   else if (xdiffDHP > x1diffDHP)
    {
relative difference between 'exact result' and 'precise result' is smaller
that relative difference between 'dco result' and 'precise result' thus
'exact result' is "better"
    m_n_exactBetter++;
    }
   else
    {
    m_n_dco_exactSame++;
    }
```

```
calculate the largest relative difference between
'dco result' and 'precise result'
  if ( xdiffDHP > m_dhpDiffdcoMax )
   {
   m_dhpDiffdcoMax = xdiffDHP;
   }
calculate the largest relative difference between
'exact result' and 'precise result'
  if ( x1diffDHP > m_dhpDiffexactMax )
   {
   m_dhpDiffexactMax = x1diffDHP;
   }

set 'd' to the relative difference between x[i] and x1[i], note that d >=
0.
  d = CalRelativeDiff ( x[i], x1[i] );

calculate the largest relative difference between
'dco result' and 'exact result'
  if ( m_dTotalMaxDiff < d )
   {
   m_dTotalMaxDiff = d;
   }

  if ( d > m_dRelativeErrorThreshold )
   {
relative difference between x[i] and x1[i] exceed the threshold
the error shall be returned

determine if the error detected is the worst error observed
    if ( m_dMaxDiff < d )
     {
collect the data about that error
     m_dMaxDiff = d;
     m_nTestNumberForMaxDiff = m_GetCrntTestNumber();
     m_nIndxInternalForMaxDiff = i;
     m_d_dco_result = x[i];
     m_d_exact_result = x1[i];
     m_dhp_precise_result = xDHP[i];
     }
report error being detected
    ret = DRT_ATTEMPTED_RUN_FAILED;
   }
  }
```

```
 return ret;

}  /*   STENCILTest::m_TestRandData   */
```

implement class member m_DumpReport() to print the collected information

```
void STENCILTest::m_DumpReport()
 {
 unsigned int i;
 dhpreal x, x1, xx, xx1;
 char s[1024];

 if ( m_dMaxDiff == .0 )
  {
  printf( "Total Max relative deviation:    %e\n",
m_dTotalMaxDiff );
  }

 else // if ( m_dMaxDiff > .0 )
  {
  printf( "Max relative deviation:    %e\n", m_dMaxDiff );
  printf( "Happen in the test case %lu at the index %u\n",
m_nTestNumberForMaxDiff, m_nIndxInternalForMaxDiff );
  printf( "\tdcoRslt=\t%.17f\n\texactRslt=\t%.17f\n",
m_nIndxInternalForMaxDiff, m_d_dco_result, m_d_exact_result );
```

get_string' being defined by the Dalsoft High Precision (dhp) package and generates inside array 's' up to 32 decimal digits of the argument

```
  m_dhp_precise_result.get_string( s, 1023, 32 );
  printf( "\tpreciseRslt=\t%s\n", s );

  printf( "\tAbsDiff_dco_exact\t%.17f\n",
          fabs( m_d_dco_result - m_d_exact_result ) );

  x = m_d_dco_result;
  x1 = m_d_exact_result;

  xx = fabs( x - m_dhp_precise_result );
  xx.get_string( s, 1023, 32 );
  printf( "\tAbsDiff_dco_precise\t%s\n", s );

  xx1 = fabs( x1 - m_dhp_precise_result );
  xx1.get_string( s, 1023, 32 );
  printf("\tAbsDiff_exact_precise\t%s\n", s);

  printf("\n");
 }

  printf("MaxRelativeDiff:\n");

  m_dhpDiffdcoMax.get_string(s, 1023, 32);
  printf("\tdco_precise\t%s\n", s);

  m_dhpDiffexactMax.get_string(s, 1023, 32);
  printf("\texact_precise\t%s\n", s);

  printf("\ndco results better in\t%lu cases\nexact results
better in\t%lu cases\nresults are the Same in\t%lu cases\n",
m_n_dcoBetter, m_n_exactBetter, m_n_dco_exactSame);

 printf("\n");
```
print data collected by DRT
```
 CDRT::m_DumpReport();

 }  /*  STENCILTest::m_DumpReport  */
```

implement function main to execute code testing

```
int main(int ac, char **av)
 {
```
create class for stencil testing
```
 STENCILTest rt;
```

process command line arguments the program was invoked with
```
 int stat;
 stat = rt.m_GetOptions(ac, av);
 if (stat == DRT_OPTION_EXIT_OK)
  {
  exit(0);
  }
 else if (stat == DRT_OPTION_EXIT_ERROR)
  {
  exit(1);
  }
```

run the test(s)
```
 rt.m_RunTest();
```

```
    exit( 0 );

   }  /*  main  */
```

creating the executable

The file ***stenciltest.cpp*** contains the above described implemented code. The following steps shall be taken to create an executable ***stenciltest***. In the following example we are using **g++** C++ compiler.

g++ -O2 -S stenciltest.cpp

compile the input file ***stenciltest.cpp*** creationg optimized assembly version of it ***stenciltest.s***

dco -i stenciltest.s -o stenciltest_dco.s -slct -parallel

process the generated file ***stenciltest.s*** by the Dalsoft's auto-parallelizer (***dco***) creating file ***stenciltest_dco.s***;

-slct command line option directs ***dco*** to only process (create parallel code for) section between '**,dco_start**' and '**.dco_end**'.

-parallel command line option directs ***dco*** to auto-parallelize (create parallel version of) the processed code

g++ -o stenciltest stenciltest_dco.s -fopenmp -ldrt -ldhp

generate executable file ***stenciltest*** out of the assembly file ***stenciltest_dco.s*** created by ***dco***.

note the library that is necessary to use during linking:

-fopenmp - to provide OpenMP routines used by ***dco*** during creation of parallel code

-ldrt - to provide functionality of the Dalsoft Random Test (***drt***) package utilized by the code

-dhp - to provide routines from the Dalsoft High Precision (***dhp***) package utilized by the code

rm stenciltest_dco.s

perform cleanup

usage of the test

Let run the created executable ***stenciltest*** for **100** seconds utilizing random number seed provided (be default) by ***drt***:

./***stenciltest*** -t 100

were

-t 100 - specifies time in seconds to run test for

The output generated is

```
Max relative deviation:    6.365827e-12
Happen in the test case 42 at the index 50274
        dcoRslt=       -0.00025070513187596
        exactRslt=     -0.00025070513187755
        preciseRslt=   -0.00025070513187722356057969993215
```

```
		AbsDiff_dco_exact	0.00000000000000160
		AbsDiff_dco_precise	0.00000000000000126511817962356743
		AbsDiff_exact_precise		0.00000000000000330827418275095 09

	MaxRelativeDiff:
		dco_precise	0.00000005343040798557957730620889
		exact_precise	0.00000005343040798557957730620889

	dco results better in	4162 cases
	exact results better in 9886 cases
	results are the Same in		7785952 cases

	Total 78, Attempted 78, Done 78 cases for 100.114 seconds error count = 71.
	Random number generator seed used 1588490732.
```

from which we conclude that the maximum relative difference between ***dco result*** and ***exact result*** is **6.365827e-12** and it happened during execution of the **42**'s test case (counting of test cases starts with **0**, thus **42** cases were executed before this condition was detected) while processing stencil member with the index **50272**; the various data about this ("worst") case is listed.

Also listed the maximum (for all the test cases attempted) relative difference between ***dco result*** and ***precise result*** and ***exact result*** and ***precise result***.

For all the test cases attempted
dco result was better (closer to ***precise result***) that ***exact result*** 4162 times
exact result was better that ***dco result*** 9886 times
dco result was the same as ***exact result*** 7785952 times

During execution 78 cases were processed and 71 "errors" were detected (the relative difference between ***dco result*** and ***exact result*** exceeded `m_dRelativeErrorThreshold` that was set to be **10e-16**). Random number generator seed used was **1588490732**.

Should we desire to recreate the worst case determined in the previous example, just execute:

```
	./stenciltest -nsi 1588490732 -s 42 -c 1
```

were

- `-nsi 1588490732` - establishes seed for the random number generator (the same as used in the previous run)
- `-s 42` - requests to skip the first **42** test cases
- `-c 1` - requests to execute one test case only

The output generated is

```
	skiping 42 cases...done.
	Max relative deviation:	6.365827e-12
	Happen in the test case 0 at the index 50274
		dcoRslt=	-0.00025070513187596
```

```
            exactRslt=          -0.00025070513187755
            preciseRslt=        -0.000250705131877223560579699993215
            AbsDiff_dco_exact        0.00000000000000160
            AbsDiff_dco_precise      0.00000000000000126511817962356743
            AbsDiff_exact_precise             0.00000000000000033082741827509509

    MaxRelativeDiff:
            dco_precise         0.00000000010812651015362232920059
            exact_precise       0.00000000010812651015362232920059

    dco results better in       81 cases
    exact results better in 436 cases
    results are the Same in            99483 cases

    Total 43, Attempted 1, Done 1 cases for 1.27241 seconds error count = 1.
    Random number generator seed used 1588490732.
```

The ability to exactly recreate a test case allows to perform a detailed analysis of the problem and even debug the code for the data that was determined to be problematic.

Table of Contents