Dalsoft's High Precision Package

Reference Manual

version 2.1
for the Linux/Windows operating systems

www.dalsoft.com/dhp.html

General

dhp - Dalsoft's High Precision package allows to perform floating point calculations with the precision higher than that supported by most of the numeric hardware. It provides C++ wrapper which allows for easy incorporation with the existing code.

dhp may be particularly useful in number of ways:

- to generate data that requires high precision - tables of the numeric constants, sensitive algorithms etc.
- to implement algorithms that may use out of range intermiduate results
- to implement algorithms that loose much of the precision
- to verify the accuracy of the particular implementation

Technical specifications

Current version:	2.1
Maximum size of the fraction:	unlimited*
Source code in:	C++
C++ wrapper:	provided
Operating system supported:	Linux, Windows
Documentation:	manual
Support:	on-line

*`unlimited' should be understood as `determined by the amount of the available memory'

Installation

The release package includes:

dhp.h – include file for Linux/Windows

linux_libdhp.a – library for Linux

windows_libdhp.lib – library for Windows.

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

```
sudo cp linux_libdhp.a /usr/local/lib64/libdhp.a
```

High Precision values

dhp uses high precision representation of the floating point numbers. This representation has the form (for non-zero values):

$$\text{“1.fraction} \times 2^{\text{exponent}}\text{„}$$

and provides 32 bit **exponent** (compare with 11 bits for IEEE double precision). For C programs, ***dhp*** supports high precision representations with the **fraction** size of 128 and 1024 bits (compare with 52 bits for IEEE double precision). For C++ programs, ***dhp*** supports high precision representations with any **fraction** size .

The fraction size if a high precision representation of the floating point number supported by ***dhp***, is called it ***precision***.

Compiling and linking with the dhp library

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

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

The procedures for linking ***dhp*** with the rest of your program vary according to the compiler and method you are using. When using C compiler, you must include the standard C++ library. For example, on a typical Linux system this can be done as following

gcc myprogram.c -ldhp -lstdc++

or

g++ myprogram.cpp -ldhp

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

Exceptions

dhp functions, in addition to producing result, may also generate error code (exceptions). The possible error codes defined in **dhp.h** file and are:

- **DHP_ERR_OK** - no errors, set to 0
- **DHP_ERR_INV** - Invalid - invalid argument
- **DHP_ERR_DZE** - ZeroDivide - divide by zero
- **DHP_ERR_OVF** - Overflow
- **DHP_ERR_UNF** - Underflow

In this document for every ***dhp*** function we list the possible exceptions it may generate. The global variable **dhp_error** contains an error code of the last ***dhp*** function executed. The global variable **dhp_error_sum** contains summary of all errors reported during the use of ***dhp***. It is a good idea to periodically check this variables to ensure they are equal to **DHP_ERR_OK**. Note that in the case of exception, the results of ***dhp*** functions are undefined. Note also that ***dhp*** items are always valid high precision floating point values.

Programming with dhp

This section describes how to make ***dhp*** calls in your program.

Global Variables

dhp_error

unsigned int dhp_error;

dhp_error contains the error code from the last executed *dhp* function. If the call succeeded, **dhp_error** will be 0 (**DHP_ERR_OK**). A list of error code values, their meanings, and defined constants for them can be found in the section **Exceptions**.

dhp_error_sum

unsigned int dhp_error_sum;

dhp_error_sum records bitwise OR of the error codes from *dhp* calls. A list of error code values, their meanings, and defined constants for them can be found in the section **Exceptions**.

dhp_version

unsigned short dhp_version;

dhp_version contains the *dhp*'s version in packed BCD format. For example, if the *dhp*'s version is 2.1, the value of **dhp_version** will be 0x0201. The high byte of **dhp_version** represents two digits to the left of the decimal point (one digit per nibble) and the low byte represents two digits to the right of the decimal point (again, one digit per nibble).

Using dhp in C++

The C++ class supported by *dhp* is **dhpreal**. It can be defined in the following ways:

dhpreal data() - defines **data** to be high precision representation of the floating point numbers that provides 32 bit exponent and 128 bits fraction.

dhpreal data(unsigned int FractionSizeInBits) - defines **data** to be high precision representation of the floating point numbers that provides provides 32 bit exponent and at least **FractionSizeInBits** bits fraction.

dhpreal data(const dhpreal& source) - defines **data** to be high precision representation identical to **source** (having the same fraction size and the same value).

Note that some compilers require explicit cast for defining **dhprel** data items with a parameter, e.g. ***dhpreal data((unsigned int)100);*** to define data to be high precision representation of the floating point numbers that provides 32 bit exponent and at least 100 bits fraction.

dhp supports all arithmetic operations on the variables of the type **dhpreal** and allows mixing them with argument of the type double. For example, the following:

```
dhpreal a, b(500);
a = 1.23;
b = a*10;
```

will define **a** to be a floating point high precision representation with the fraction size of 128 bits, **b** to be a floating point high precision representation with the fraction size of at least 500 bits, set **a** to 1.23 and **b** to 12.3.

dhp provides the following C++ class functions

get_double

double get_double()

return the value of **this** as a double.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

Example:

```
dhpreal a;
double d;

.

d = a.get_double();
```

get_string

int get_string(char *str, unsigned int strSz, unsigned int cntAfterDot)

set **str** to represent the value of **this** in the following format: **[-+][D1][.D2]** where D1 - is a decimal integer denoting the integer part of the floating point value, D2 - is a decimal integer denoting the fractional part of the floating point value. At most **strSz** characters of **str** are affected (including null character terminating the string) and at most **cntAfterDot** characters are set after the decimal point '.'.

return not-zero value if **str** contains the integer part D1 (and decimal point in the case this is not a whole number), zero – otherwise

Possible Exceptions: none

Example:

```
dhpreal a;
char s[1024];
int stat;

.

stat = a.get_string( s, 1024, 4 );
```

set_string

void set_string(const char *str)

set **this** to floating point number represented as a string **str**. Input string **str** has the following format: [-+][D1][.D2] where D1 - is a decimal integer denoting the integer part of the floating point value, D2 - is a decimal integer denoting the fractional part of the floating point value. Result of on inputs not confirming to the above description is undefined.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

Example:

```
dhpreal a;

.

a.set_string( "-123.456" );
```

cast

void cast(unsigned int FractionSizeInBits)

set **this** to be high precision representation of the floating point numbers that provides provides 32 bit exponent and at least **FractionSizeInBits** bits fraction. The old value of **this** is attempted to be preserved.

Possible Exceptions: none

Example:

```
dhpreal a;
```

.

a.cast(1024);

is0

int is0()

return not-zero value if **this** is equal to 0, zero - if ***this*** has numeric representation of a not zero value.

Possible Exceptions: none

Example:

dhpreal a(2000);

.

if (!a.is0())

.

operator=

dhpreal& operator(double src)

set **this** to the value of **src**.

Possible Exceptions: D**HP_ERR_INV**, **DHP_ERR_OVF**

Example:

dhpreal a(250);
double d;

.

a = d;

dhpreal& operator=(const dhpreal& src)

set **this** to the value of **src**; the precision of **this** (the size of it fraction) is not affected.

Possible Exceptions: none

Example:

dhpreal a(250), b;

.

a = b;

dhp provides the following C++ external functions

operator+

const dhpreal operator+(const dhpreal& src1, const dhpreal& src2)

return **dhpreal** value equal **src1** + **src2**. The precision of the returned result is set to the maximum of precisions of **src1** and **src2**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

const dhpreal operator+(double src1, const dhpreal& src2)
const dhpreal operator+(const dhpreal& src1, double src2)

return **dhpreal** value equal **src1** + **src2**. The precision of the returned result is set to the maximum of precisions of the **dhpreal** arguments.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**, **DHP_ERR_INV**

Example:

dhpreal a, b(200), c;

.

a = b + c;
c = a + 1.2;

operator-

const dhpreal operator-(const dhpreal& src1, const dhpreal& src2)

return **dhpreal** value equal **src1** - **src2**. The precision of the returned result is set to the maximum of precisions of **src1** and **src2**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

const dhpreal operator-(double src1, const dhpreal& src2)
const dhpreal operator-(const dhpreal& src1, double src2)

return **dhpreal** value equal **src1** - **src2**. The precision of the returned result is set to the maximum of precisions of the **dhpreal** arguments.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**, **DHP_ERR_INV**

Example:

dhpreal a, b(200), c;

.

a = b – c;
c = a – 1.2;

operator*

const dhpreal operator*(const dhpreal& src1, const dhpreal& src2)

return **dhpreal** value equal **src1** * **src2**. The precision of the returned result is set to the maximum of precisions of **src1** and **src2**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

const dhpreal operator*(double src1, const dhpreal& src2)
const dhpreal operator*(const dhpreal& src1, double src2)

return **dhpreal** value equal **src1** * **src2**. The precision of the returned result is set to the maximum of precisions of the **dhpreal** arguments.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**, **DHP_ERR_INV**

Example:

dhpreal a, b(200), c;

.

a = b * c;
b = c*3.4;

operator/

const dhpreal operator/(const dhpreal& src1, const dhpreal& src2)

return **dhpreal** value equal **src1** / **src2**. The precision of the returned result is set to the maximum of precisions of **src1** and **src2**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**, **DHP_ERR_DZE**

const dhpreal operator/(double src1, const dhpreal& src2)
const dhpreal operator/(const dhpreal& src1, double src2)

return **dhpreal** value equal **src1** / **src2**. The precision of the returned result is set to the maximum of precisions of the **dhpreal** arguments.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**, **DHP_ERR_INV**, **DHP_ERR_DZE**

Example:

dhpreal a, b(200), c;

.

a = b / c;
c = 5.6/b;

compare functions

const int operator<(const dhpreal& src1, const dhpreal& src2)
const int operator<=(const dhpreal& src1, const dhpreal& src2)
const int operator>(const dhpreal& src1, const dhpreal& src2)
const int operator>=(const dhpreal& src1, const dhpreal& src2)
const int operator==(const dhpreal& src1, const dhpreal& src2)
const int operator!=(const dhpreal& src1, const dhpreal& src2)

Possible Exceptions: none

const int operator<(const dhpreal& src1, double src2)
const int operator<(double src1, const dhpreal& src2)
const int operator<=(const dhpreal& src1, double src2)
const int operator<=(double src1, const dhpreal& src2)
const int operator>(const dhpreal& src1, double src2)
const int operator>(double src1, const dhpreal& src2)
const int operator>=(const dhpreal& src1, double src2)
const int operator>=(double src1, const dhpreal& src2)
const int operator==(const dhpreal& src1, double src2)
const int operator==(double src1, const dhpreal& src2)
const int operator!=(const dhpreal& src1, double src2)
const int operator!=(double src1, const dhpreal& src2)

return not-zero value if corresponding relation between **src1** and **src2** is true, zero – otherwise.

Possible Exceptions: **DHP_ERR_INV**

Example:

dhpreal a, b(200), c;

.

if ((a < b) && (c >= 2.3))

sqrt

dhpreal sqrt(const dhpreal& src)

return **dhpreal** value equal SQRT(**src**). The precision of the returned result is set to the precision of **src**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**, **DHP_ERR_INV**

Example:

dhpreal a, b(200);

.

a = sqrt(b);

trunc

dhpreal trunc(const dhpreal& src)

return **dhpreal** value equal to the integer nearest to **src** and not larger in absolute value than **src**. The precision of the returned result is set to the precision of **src**.

Possible Exceptions: none

Example:

dhpreal a, b(200);

.

a = trunc(b);

round

dhpreal round(const dhpreal& src)

return **dhpreal** value equal to the integer nearest to **src**. The precision of the returned result is set to the precision of **src**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

Example:

dhpreal a, b(200);

.

a = round(b);

fabs

dhpreal fabs(const dhpreal& src)

return **dhpreal** value equal to the absolute value of **src**. The precision of the returned result is set to the precision of **src**.

Possible Exceptions: none

Example:

dhpreal a, b(200);

.

a = fabs(b);

cmp

int cmp(const dhpreal& src1, const dhpreal& src2)

return an integer less than, equal to, or greater than zero is **src1** is , respectively, less than, equal, or greater than **src2**

Possible Exceptions: none

Example:

dhpreal a, b(200);

.

if (cmp(a, b) < 0)

Using dhp in C

All functions, provided by ***dhp*** have name that begins with '*dhp_*' or '*dhp1024_*' (depending on the type of it arguments/result) and followed by the low-case mnemonic, e.g '*dhp_add*'.

The C data types supported by ***dhp*** are **dhp_t** and **dhp1024_t**

dhp_t is a high precision representation of the floating point numbers that provides 32 bit exponent and 128 bits fraction.

dhp1024_t is a high precision representation of the floating point numbers that provides 32 bit exponent and 1024 bits fraction.

dhp provides several sets of functions for each of the supported data types.

1. Functions for copying data to and from formats supported by the ***dhp***. These functions are:
    - **dhp_set_double**, **dhp_get_double**, **dhp_set_string**, **dhp_get_string**
    - **dhp1024_set_double**, **dhp1024_get_double**, **dhp1024_set_string**, **dhp1024_get_string**
2. Functions which are intended to be orthogonal with the standard floating point operations (+, -, *,/ etc.). These functions are:
    - **dhp_add**, **dhp_sub**, **dhp_mul**, **dhp_div**, **dhp_sqrt**, **dhp_trunc**, **dhp_round**, **dhp_fabs**, **dhp_neg**
    - **dhp1024_add**, **dhp1024_sub**, **dhp1024_mul**, **dhp1024_div**, **dhp1024_sqrt**, **dhp1024_trunc**, **dhp1024_round**, **dhp1024_fabs**, **dhp1024_neg**
3. Comparison functions:
    - **dhp_is0**, **dhp_cmp**
    - **dhp1024_is0**, **dhp1024_cmp**

dhp_get_double, dhp1024_get_double

double dhp_get_double(dhp_t src)
double dhp1024_get_double(dhp1024_t src)

return the value of **src** as a double.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

Example:

dhp_t a; double d;

.

d = dhp_get_double(a);

dhp_get_string, dhp1024_get_string

int dhp_get_string(dhp_t, src, char *str, unsigned int strSz, unsigned int cntAfterDot)
int dhp1024_get_string(dhp1024_t, src, char *str, unsigned int strSz,unsigned int cntAfterDot)

set **str** to represent the value of **src** in the following format:
[-+][D1][.D2]
where D1 - is a decimal integer denoting the integer part of the floating point value D2 - is a decimal integer denoting the fractional part of the floating point value. At most **strSz** characters of **str** are affected (including null character terminating the string) and at most **cntAfterDot** characters are set after the decimal point '.'.

return not-zero value if **str** contains the integer part D1 (and decimal point in the case this is not a whole number), zero - otherwise

Possible Exceptions: none

Example:

dhp_t a; char s[1024]; int stat;

.

stat = dhp_get_string(a, s, 1024, 4);

dhp_set_double, dhp1024_set_double

dhp_t dhp_set_double(double src)
dhp1024_t dhp1024_set_double(double src)

return the item of an appropriate data type that has high precision representation of **src**.

Possible Exceptions: **DHP_ERR_INV**, **DHP_ERR_OVF**

Example:

dhp1024_t a; double d;

.

a = dhp1024_set_double(d);

dhp_set_string, dhp1024_set_string

dhp_t dhp_set_string(const char *str)
dhp1024_t dhp1024_set_string(const char *str)

return the item of an appropriate data type that has high precision floating point number with the value represented as a string **str**. Input string str has the following format:
[-+][D1][.D2]

where D1 - is a decimal integer denoting the integer part of the floating point value D2 - is a decimal integer denoting the fractional part of the floating point value. Result of on inputs not confirming to the above description is undefined.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

Example:

dhp_t a;

.

a = dhp_set_string("-123.456");

dhp_add, dhp1024_add

dhp_t dhp_add(dhp_t src1, dhp_t src2)
dhp1024_t dhp1024_add(dhp1024_t src1, dhp1024_t src2)

return the item of an appropriate data type that has high precision floating point number with the value equal to **src1** + **src2**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

Example:

dhp1024_t a, b, c;

.

a = dhp1024_add(b, c);

dhp_sub, dhp1024_sub

dhp_t dhp_sub(dhp_t src1, dhp_t src2)
dhp1024_t dhp1024_sub(dhp1024_t src1, dhp1024_t src2)

return the item of an appropriate data type that has high precision floating point number with the value equal to **src1** - **src2**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

Example:

dhp_t a, b, c;

.

a = dhp_sub(b, c);

dhp_mul, dhp1024_mul

dhp_t dhp_mul(dhp_t src1, dhp_t src2)
dhp1024_t dhp1024_mul(dhp1024_t src1, dhp1024_t src2)

return the item of an appropriate data type that has high precision floating point number with the value equal to **src1** * **src2**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**

Example:

dhp_t a, b, c;

.

a = dhp_mul(b, c);

dhp_div, dhp1024_div

dhp_t dhp_div(dhp_t src1, dhp_t src2)
dhp1024_t dhp1024_div(dhp1024_t src1, dhp1024_t src2)

return the item of an appropriate data type that has high precision floating point number with the value equal to **src1** / **src2**.

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**, **DHP_ERR_DZE**

Example:

```
dhp_t a, b, c;

.

a = dhp_div( b, c );
```

dhp_sqrt, dhp1024_sqrt

dhp_t dhp_sqrt(dhp_t src)
dhp1024_t dhp1024_sqrt(dhp1024_t src)

return the item of an appropriate data type that has high precision floating point number with the value equal to SQRT(**src**)

Possible Exceptions: **DHP_ERR_OVF**, **DHP_ERR_UNF**, **DHP_ERR_INV**

Example:

```
dhp_t a, b;

.

a = dhp_sqrt( b );
```

dhp_trunc, dhp1024_trunc

dhp_t dhp_trunc(dhp_t src)
dhp_t dhp1024_trunc(dhp1024_t src)

return the item of an appropriate data type that has high precision floating point number with the value equal to the integer nearest to **src** and not larger in absolute value than **src**.

Possible Exceptions: none

Example:

```
dhp_t a, b;

.

a = dhp_trunc( b );
```

dhp_round, dhp1024_round

dhp_t dhp_round(dhp_t src)
dhp_t dhp1024_round(dhp1024_t src)

return the item of an appropriate data type that has high precision floating point number with the value equal to the integer nearest to **src**.

Possible Exceptions: none

Example:

```
dhp_t a, b;

.

a = dhp_trunc( b );
```

dhp_fabs, dhp1024_fabs

dhp_t dhp_fabs(dhp_t src)
dhp1024_t dhp1024_fabs(dhp1024_t src)

return the item of an appropriate data type that has high precision floating point number with the value equal the absolute value of **src**.

Possible Exceptions: none

Example:

```
dhp_t a, b;

.

a = dhp_fabs( b );
```

dhp_neg, dhp1024_neg

dhp_t dhp_neg(dhp_t src)
dhp1024_t dhp1024_neg(dhp1024_t src)

return the item of an appropriate data type that has high precision floating point number with the value equal to -**src**.

Possible Exceptions: none

Example:

```
dhp_t a, b;
```

.

a = dhp_neg(b);

dhp_is0, dhp1024_is0

int dhp_is0(dhp_t src)
int dhp1024_is0(dhp1024_t src)

return not-zero value if **src** is equal to 0, zero - if **src** has numeric representation of a not zero value.

Possible Exceptions: none

Example:

dhp_t a;

.

if (!dhp_is0(a))

.

dhp_cmp, dhp1024_cmp

int dhp_cmp(dhp_t src1, dhp_t src2)
int dhp1024_cmp(dhp_t src1, dhp_t src2)

return and integer less than, equal to, or greater than zero is **src1** is, respectively, less than, equal, or greater than **src2**

Possible Exceptions: none

Example:

dhp_t a, b;

.

if (dhp_cmp(a, b) < 0)

Examples

computing the sequence of Fibonacci numbers

Assume that the task is to compute the first 100 items from the sequence of Fibonacci numbers

$$F_0 = 0,\ F_1 = 1,\ F_{n+2} = F_{n+1} + F_n$$

using de Moivre's-Binet's formula

$$F_n = (\varphi^n - \varphi'^n)/ \sqrt{5}$$

where

$$\varphi = (1 + \sqrt{5})/2,\ \varphi' = (1 - \sqrt{5})/2$$

The following program uses standard double precision variables to achieve this task:

```
double p, p1, pn, p1n, fn, sqrt5;
int i;

sqrt5 = sqrt( 5. );
p = ( 1. + sqrt5 )/2.;
p1 = ( 1. - sqrt5 )/2.;
pn = p;
p1n = p1;
for ( i = ; i < 100; i++ ) {
        fn = ( pn - p1n )/sqrt5;
        printf( "F%d = %f\n", i, fn );
        pn *= p;
        p1n *= p1;
}
```

using dhp in C

In C the above program can be rewritten to utilize ***dhp*** as following:

```
#include <dhp.h>
.
dhp_t p, p1, pn, p1n, fn, sqrt5;
int i;
char s[1024];

sqrt5 = dhp_sqrt( dhp_set_double( 5. ) );
```

```
p = dhp_div( dhp_add( dhp_set_double( 1. ), sqrt5 ), dhp_set_double( 2. ) );
p1 = dhp_div( dhp_add( dhp_set_double( 1. ), sqrt5 ), dhp_set_double( 2. ) );
pn = p;
p1n = p1;
for ( i = ; i < 100; i++ ) {
        fn = dhp_div( dhp_sub( pn, p1n ), sqrt5 );
        dhp_get_string( fn, s, 1000, 6 );
        printf( "F%d = %s\n", i, s );
        pn = dhp_mul( pn, p );
        p1n = dhp_mul( p1n, p1 );
}
```

This example shows that implementing ***dhp*** functionality in C requires significant modification of the original program.

using dhp in C++

The program that uses standard double precision variables (listed above) can be rewritten to utilize ***dhp*** as following:

```
#include <dhp.h>
.
dhpreal p, p1, pn, p1n, fn, sqrt5;
int i;
char s[1000];

fn = 5.;
sqrt5 = sqrt( fn );
p = ( 1. + sqrt5 )/2.;
p1 = ( 1. - sqrt5 )/2.;
pn = p;
p1n = p1;
for ( i = ; i < 100; i++ ) {
        fn = ( pn - p1n )/sqrt5;
        fn.get_string( s, 1000, 6 );
        printf( "F%d = %s\n", i, s );
        pn *= p;
        p1n *= p1;
}
```

The changes from the original (normal) program are highlighted in this color. This example shows that implementing ***dhp*** functionality in C++ is straightforward and

requires very minor modification of the original program.

Results of execution

The following table lists generated Fibonacci number for selected indexes:

1. The column under **#** presents the index of Fibonacci number in the following cells of the raw.
2. The columns under **double** header presents Fibonacci numbers generated by the program using standard double precision variables (listed above).
3. The columns under **64-bit uints** header presents Fibonacci numbers generated by the program utilizing the original formula using 64-bit unsigned integer variables.
4. The columns under **dhp** header presents Fibonacci numbers generated by the program using ***dhp*** (listed above, C and C++ versions generated the identical results).

Results that are not accurate marked in this color.

#	double	64-bit uints	dhp
68	72723460248141.156250	72723460248141	72723460248141.000000
69	117669030460994.250000	117669030460994	117669030460994.000000
70	190392490709135.406250	190392490709135	190392490709135.000000
71	308061521170129.625000	308061521170129	308061521170129.000000
72	498454011879265.125000	498454011879264	498454011879264.000000
73	806515533049394.750000	806515533049393	806515533049393.000000
74	1304969544928659.750000	1304969544928657	1304969544928657.000000
75	2111485077978054.750000	2111485077978050	2111485077978050.000000
76	3416454622906715.000000	3416454622906707	3416454622906707.000000
77	5527939700884770.000000	5527939700884757	5527939700884757.000000
78	8944394323791484.000000	8944394323791464	8944394323791464.000000
79	14472334024676254.000000	14472334024676221	14472334024676221.000000
80	23416728348467736.000000	23416728348467685	23416728348467685.000000
81	37889062373143992.000000	37889062373143906	37889062373143906.000000
82	61305790721611736.000000	61305790721611591	61305790721611591.000000
83	99194853094755728.000000	99194853094755497	99194853094755497.000000
84	160500643816367456.000000	160500643816367088	160500643816367088.000000
85	259695496911123200.000000	259695496911122585	259695496911122585.000000
86	420196140727490688.000000	420196140727489673	420196140727489673.000000
87	679891637638613888.000000	679891637638612258	679891637638612258.000000
88	1100087778366104576.000000	1100087778366101931	1100087778366101931.000000
89	1779979416004718592.000000	1779979416004714189	1779979416004714189.000000
90	2880067194370823680.000000	2880067194370816120	2880067194370816120.000000
91	4660046610375542784.000000	4660046610375530309	4660046610375530309.000000
92	7540113804746366976.000000	7540113804746346429	7540113804746346429.000000
93	12200160415121909760.000000	12200160415121876738	12200160415121876738.000000
94	19740274219868274688.000000	1293530146158671551	19740274219868223167.000000
95	31940434634990190592.000000	13493690561280548289	31940434634990099905.000000
96	51680708854858465280.000000	14787220707439219840	51680708854858323072.000000
97	83621143489848655872.000000	9834167195010216513	83621143489848422977.000000

98	135301852344707137536.000000	6174643828739884737	135301852344706746049.000000
99	2189229958345557934 08.000000	16008811023750101250	2189229958345551690 26.000000

Rump's Example

"Rump's example" is to compute the expression

$$f = (333.75 - a^2)b^6 + a^2 (11a^2 b^2 - 121b^4 - 2) + 5.5b^8 + a/(2b)$$

with $a = 77617$ and $b = 33096$.

Attempts to implement the above formula using the standard (single or double or extended) precision, available on the modern computers, produces wrong result for f.

The following program uses standard double precision variables to implement "Rump's example":

```
double a, b, a2, b2, b4, b6, b8, f;

a = 77617.;
b = 33096.;
a2 = a*a;
b2 = b*b;
b4 = b2*b2;
b6 = b4*b2;
b8 = b4*b4;

f = ( 333.75 - a2 )*b6 + a2* ( 11.*a2*b2 - 121.*b4 -2. ) + 5.5*b8 + a/(2.*b);

printf( "IEEE\t%f\n", f );
```

Execution pf this program generates (the wrong) result:

```
IEEE   1.172604
```

The program that uses standard double precision variables (listed above) can be rewritten to utilize ***dhp*** as following:

```
#include <dhp.h>
.
dhpreal a, b, a2, b2, b4, b6, b8, f;
char s[1024];

a = 77617.;
```

```
b = 33096.;
a2 = a*a;
b2 = b*b;
b4 = b2*b2;
b6 = b4*b2;
b8 = b4*b4;

f = ( 333.75 - a2 )*b6 + a2* ( 11.*a2*b2 - 121.*b4 -2. ) + 5.5*b8 + a/(2.*b);

f.get_string( s, 1023, 64 );
printf( "dhp\t%s\n", s );
```

Execution of this program generates (the correct) result:

dhp -.827396059946821368141165095479816291998435849860906829020606651 1

The changes from the original (normal) program are highlighted in this color. This example shows that implementing ***dhp*** functionality in C++ is straightforward and requires very minor modification of the original program. Note also how easy it is, using ***dhp***, to print the data with “any” desirable precision.

Using to test parallelization of a stencil

See this on how ***dhp*** was used to test parallel code generated by Dalsoft Code Optimizer (***dco***).

Table of Contents