Algorithm to compute π

Question 1:

Construct an algorithm in C++ to approximate π. Assume that you are working on a project with a small development team and your algorithm will be used as a part of a larger code base.

- Assume that the required precision of the calculation is not known until run-time.
- You do not need to determine the error $|x - \pi|$ associated with your calculation.
- Your algorithm should be able to work with single or double-precision floating point numbers
- Assume you have access to a compiler that supports any version of C and/or C++ that you would like numbers.

You can use any method, libraries etc. that you would like with the exception of functions that approximate transcendental numbers (e.g. std::asin, M_PI or similar).

It might be useful to know that $\pi = 4\sum_{k=1}^{\infty}\frac{(-1)^{k+1}}{2k-1}$ (known as the Gregory Series). The first few terms in the series are $4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots)$.

This is not the only way to compute π and there is no special reason why this method should be preferred over any other method.

You do need to write a Make file, compiler/linker commands or explicitly write out #include statements, but you are responsible for appropriately using namespaces.

Possible Answer 1:

Solution using Gregory Series:

Start with the identity $\pi \approx 4\sum_{k=0}^{N}\frac{(-1)^{k+1}}{2k-1}$

```
\* run_time_pi.h *\

namespace wbp {

template< typename T >
T partial_sum( T (*f)(uint index), uint sum_start, uint sum_end ) {

   T sum_total = static_cast<T>( 0 );

   for( uint i = sum_start; i <= sum_end ; i++ ) {
       sum_total += f(i);
   }

   return sum_total;

}

template < typename T >
```

```
T alt_sign( uint k ) {
   return ( ( k + 1 )%2 == 0 )?( static_cast<T>( 1 ) ):( static_cast<T>( -1 ) );
}

template < typename T >
T gregory_term( uint k ) {

   return alt_sign<T>( k )*static_cast<T>(1) / static_cast<T>( 2*k - 1 );
}

template< typename T >
T gregory_pi( uint num_iter ) {

  static_assert( std::is_floating_point<T>::value,
   "Template parameter must be a floating-point value.");
  return static_cast<T>( 4 )*partial_sum( &gregory_term<T>, 1, num_iter );

}

}
```

Solution using Power Series:

Start with the identity $\pi \approx \sqrt{6 \sum_{k=1}^{N} \frac{1}{k^2}}$

```
\* run_time_pi.h *\

namespace wbp {

template< typename T >
T partial_sum( T (*f)(uint index), uint sum_start, uint sum_end ) {

   T sum_total = static_cast<T>( 0 );

   for( uint i = sum_start; i <= sum_end ; i++ ) {
       sum_total += f(i);
   }

   return sum_total;

}

template < typename T >
T pow_term( uint k ) {

    return static_cast<T>( 1 )/static_cast<T>( k * k );

```

```
}

template< typename T >
T power_pi( uint num_iter ) {

    static_assert( std::is_floating_point<T>::value,
    "Template parameter must be a floating-point value.");
    return std::sqrt( static_cast<T>( 6 )*partial_sum( &pow_term<T>, 1, num_iter ) );

}

}
```

Question 2:

Suppose your team notices that your algorithm is a performance bottle-neck, how would you speed it up?

Possible Answer 2:

For any of these series methods the terms in the series are independent, as they only depend on index. The series terms can be computed in a canonical "for" loop and easily parallelized using OpenMP for similar.

For a method like the Monte Carlo method each point can be assigned a position independently of one another.

Question 3:

Suppose someone else in your team has written an algorithm to compute π using the Gregory Series $\left(\pi = 4\sum_{k=0}^{\infty} \frac{(-1)^{k+1}}{2k-1}\right)$.

Their algorithm, in pseudo-code, takes the following form:

Algorithm 1 Estimate π using Gregory Sum formula, using pairs of terms.

Input: $n \in \mathbb{N}$ ▷ Specify the number of terms in the series – must be an even number

1: $x \leftarrow 0$ ▷ Initialize the total sum to zero.
2: **for** $k \in [1, 3, 5, \ldots, n/2 - 1]$ **do**
3: $\quad x \leftarrow x + \left(\frac{1}{2k-1} - \frac{1}{2(k+1)-1}\right)$
4: **end for**

return $4x$

The first few terms using this algorithm would read $4((1 - \frac{1}{3}) + (\frac{1}{5} - \frac{1}{7}) + \ldots)$.

You notice that after a large number of iterations this algorithm exhibits some strange behavior – it does not seem to converge to π. You decide to investigate further and notice the error term $|x - \pi|$ does not go to zero. What might be causing this problem?

Possible Answer 3:

As k grows large we will find that $\frac{1}{2k-1} \approx \frac{1}{2(k+1)-1} \leftrightarrow \frac{1}{2k-1} - \frac{1}{2(k+1)-1} \approx 0$. We also expect that these terms will have similar numbers of significant digits – this is a situation that will almost certainly result in loss of significance / catastrophic cancellation. The result of this (might) be the error term converging to some non-zero constant, or diverging.

Question 4:

Suppose your team decides that the number of iterations/recursions necessary to guarantee required precision can be determined at compile time. Further it has been determined that double-precision floating-point numbers are the only type that needs to be supported.

How would you re-write your algorithm to be computed at compile time instead of run time?

Possible Answer 4:

Possible answer if starting with the Gregory Sum is stated below – answer for other series methods are similar.

```
\* meta_pi.h *\
namespace wbp {

// General case
template<unsigned int N, typename T = double>
struct GregorySum {
    static constexpr T sum =
     ( (N % 2) ? 1.0 : -1.0 ) / ( 2*N - 1 ) +  GregorySum<N - 1, T>::sum;
};

// Specialized stop case
template<typename T>
struct GregorySum<1, T> {
    static constexpr T sum = 1;
};

template< unsigned int N >
double compute_meta_pi() {
    return 4.0*GregorySum< N >::sum;
}

}
```

Usage: `wbp::compute_meta_pi<101>()`