Instituto Tecnológico de Costa Rica
Área de Ingeniería en Computadores
Profesor: Dr. Pablo Alvarado Moya
CE-3102 Análisis Numérico para Ingeniería — Fecha de entrega: **31 de agosto, 2018**
II Semestre 2018

Proyecto 1

1 Introducción

En este proyecto usted implementará métodos para encontrar todas las raíces de polinomios, reales y complejas. Los coeficientes de los polinomios pueden tener tipos `float`, `double`, `std::complex<float>` o `std::complex<double>`. Por ello, los métodos que usted implementará deberán utilizar plantillas (*C++ templates*) y estrategias de metaprogramación que permitan la selección de segmentos de código especializados en cada caso.
La biblioteca `boost` ofrece gran cantidad de estructuras de datos y algoritmos para utilizar con el lenguaje de programación C++. Mucho del código de `boost` ha pasado a ser parte de las bibliotecas estándar de C++ (STL). En este proyecto usted utilizará de la clase `boost::math::tools::polynomial`, que ofrece una forma sencilla de representar y manipular polinomios.
El método externo `evaluate_polynomial` permite evaluar el polinomio en un punto dado. `Boost` también ofrece funcionalidad básica para realizar operaciones aritméticas polinomiales, aunque los autores advierten que los métodos no están optimizados.
Usted además utilizará la biblioteca `program_options` de `boost` para interpretar los argumentos de la línea de comandos.

2 Objetivos

2.1 Objetivo general

Implementar y evaluar algoritmos para búsqueda de raíces de polinomios, capaces de encontrar todas las raíces reales y complejas de un polinomio.

2.2 Objetivos específicos

1. Implementar un algoritmo de deflación polinomial
2. Implementar el método de Müller para búsqueda de raíces
3. Utilizar deflación polinomial para buscar las raíces.
4. Utilizar pulido de raíces para mejorar los resultados del algoritmo
5. Implementar el método de Jenkins-Traub para búsqueda de raíces
6. Evaluar el desempeño de los algoritmos implementados

3 Procedimiento

1. Revise el método de evaluación de polinomios implementado en `boost`. ¿Qué tipo de algoritmo utilizan?

2. Revise el código base entregado para la realización del proyecto. El proceso de compilación es configurado con CMake y el destino de los ejecutables queda en el directorio `build/bin`. Revise el archivo `README.txt` para más detalles.

3. Implemente en el archivo `include/Deflation.hpp` una función externa para realizar la deflación polinomial, dada una raíz. La función recibe un polinomio de entrada y la raíz que se quiere deflacionar, y produce un nuevo polinomio con el cociende a la salida, y el residuo de la división como un tercer argumento pasado por referencia:

```
namespace anpi {
  namespace bmt=boost::math::tools; // bmt as alias for boost::math::tools

  /**
   * Deflate polynomial
   *
   * @param[in] poly Input polynomial to be deflated with the provided root
   * @param[in] root Root of poly to be deflated with.
   * @param[out] residuo Residual of the polynomial deflation
   * @return deflated polynomial
   */
  template<class T>
  bmt::polynomial<T> deflate(const bmt::polynomial<T>& poly,
                             const T& root,
                             T& residuo,
                             T& tolerance=anpi::epsilon<T>());
}
```

   Usted debe implementar la deflación como caso particular de la división polinomial por un polinomio de primer orden. Por tanto, no debe utilizar la división polinomial que ya le ofrece `boost` para implementar la deflación, pues la división es un método de mayor complejidad.

   Implemente pruebas unitarias para verificar el funcionamiento correcto de su método, asegurándose de comprobar casos extremos.

4. Para el caso en que el polinomio de entrada solo tenga coeficientes reales, y la raíz sea compleja, implemente otra función externa que realice la deflación polinomial pero para el par de raíces complejas conjugadas, es decir, el método debe eliminar el par de raíces complejas conjugadas de una sola vez.

```
template<class T>
bmt::polynomial<T> deflate2(const bmt::polynomial<T>& poly,
                            const std::complex<T>& root,
                            bmt::polynomial<T>& residuo,
                            T& tolerance=anpi::epsilon<T>())
```

   Implemente pruebas unitarias para verificar el funcionamiento correcto de su método, asegurándose de comprobar casos extremos.

5. Implemente en el archivo `include/Muller.hpp` el método de Müller para buscar todas las raíces de un polinomio. Su método debe permitir indicar un punto inicial

para buscar las raíces, y una bandera para indicar si se desea aplicar pulido o no de las raíces:

```
namespace anpi {
  /// Enum to make polish explicit
  enum PolishEnum {
    DoNotPolish,
    PolishRoots
  };

  /**
   * Compute the roots of the given polynomial using the Muller method.
   * @param[in] poly polynomial to be analyzed for roots
   * @param[out] roots all roots found
   * @param[in] start initial point for finding the roots
   * @param[in] polish indicate if polishing is needed or not.
   *
   * @return the number of roots found
   */
  template<class T, class U>
  void muller(const bmt::polynomial<T>& poly,
                    std::vector<U>& roots,
              const PolishEnum polish=DoNotPolish,
              const U start=U()) {
    static_assert(std::is_floating_point<T>::value ||
                  boost::is_complex<T>:value,
                  "T must be floating point or complex");
    static_assert(std::is_floating_point<U>::value ||
                  boost::is_complex<U>:value,
                  "U must be floating point or complex");
  }
}
```

Deben encontrarse todas las raíces, por lo que se debe utilizar la deflación polinomial implementada en los puntos anteriores. Puede recurrir al libro "Numérical Recipes" de Press et al. para más detalles.

Su método debe permitir activar y desactivar el pulido de raíces para que usted pueda comparar la diferencia.

Implemente pruebas unitarias para verificar el funcionamiento correcto de su método, asegurándose de comprobar casos extremos y distintas combinaciones de tipos de coeficientes y de raíces.

Observe que el código base entregado con este enunciado ya se encarga de presentar los resultados en un formato concreto que no debe ser variado, y también le permite recibir por línea de comando el polinomio y con las distintas opciones de uso. Revise el archivo `src/main.cpp` para más detalles.

En los puntos listados más abajo se brindan detalles adicionales que debe tomar en cuenta en esta implementación.

6. Investigue e implemente el método Jenkins-Traub y compare la convergencia en la búsqueda de todas las raíces de un polinomio contra el método de Muller implementado en el punto anterior.

7. El código base disponible para este proyecto ya ofrece en el programa principal una interfaz básica por línea de comandos que permite combinar los tipos utilizados para coeficientes y raíces, seleccionar los métodos, y activar el pulido.

Compile dicho código y revise la salida de la opción `--help`.

Usted debe agregar una opción `--start` (o `-s`) para indicar el punto de inicio para el método de Müller. Ya el código disponible es capaz de pasar esa información al método (revise el atributo `start` de la estructura `Config`).

8. Asegúrese de que sus funciones puedan manejar correctamente las distintas posibilidades de tipos para los coeficientes y para las raíces.

   Sus métodos deben poder manejar raíces complejas, y por tanto las plantillas deben aceptar, además de los tipos de punto flotante que ofrece C++, los tipos `std::complex<float>` y `std::complex<double>` para los coeficientes y para las raíces.

   Si su función retorna raíces con tipos `float` o `double`, puede ser usado solo para encontrar raíces reales, y en caso de que el algoritmo detecte que hay raíces complejas, simplemente debe descartarlas con la deflación correspondiente, y reportar únicamente las raíces reales.

   Revise los métodos utilizados en el código diponible para convertir las cadenas de caracteres a los polinomios, y viceversa, pues allí ya se usan técnicas de metaprogramación para hacer ese tipo de decisiones en tiempo de compilación.

9. Diseñe un experimento para evaluar cómo se degrada la precisión de las raíces encontradas en términos del orden del polinomio, de la precisión utilizada (`float` o `double`) y de si las raíces son o no complejas. Usted debe decidir y justificar qué tipo de error utilizar para medir la precisión de las raíces. Esto es tema central en el artículo a presentar. Recuerde que en dicho artículo nunca debe presentar como resultados "pantallazos" (revisar guía para artículos).

10. Evalúe el desempeño de su implementación en tiempo de ejecución y número de operaciónes básicas empleadas. Para ello debe utilizar un perfilador (gprof, valgrind/callgrind, oprofile, etc.). También los resultados correspondientes a esto debe presentarlos en el artículo debidamente tabulados.

11. Realice un artículo formal, donde presente los resultados obtenidos. El esquema de un artículo formal lo puede encontrar en el sitio del curso, en donde el énfasis debe darlo a la evaluación de los métodos, explicando brevemente los métodos utilizados. Dicho artículo no debe exceder las 3 páginas.

4 Entregables

El código fuente y el artículo científico deben ser entregados según lo estipulado en el programa del curso.
Incluya un archivo de texto README con las instrucciones para compilación y ejecución del programa.
Su código será evaluado por medio de pruebas de sistema que verificarán la funcionalidad de todas las opciones. Esto es, se utilizará un script que ejecuta varias pruebas con distintos casos de uso con todas las combinaciones de tipos y verifica que las salidas

correspondan a lo esperado. Por ejemplo, si se piden raíces de tipo `double`, entonces si el polinomio tiene raíces complejas, deben retornarse las raíces reales, etc.