ADVANCED MATHEMATICAL PROGRAMMING

MTH399 / U15161 – Level 3

UNIVERSITY OF PORTSMOUTH

Spring Semester Coursework, Part II

Academic Session: 2010–2011

INSTRUCTIONS

a) **Deadline: Monday, May 16, 2010 (before 4PM) on Victory**

b) This assignment makes up **60% of the unit assessment**.

c) Complete **all** Parts, 1, 2 and 3.

d) Upload all code files to the Victory assignment; no printed copies are needed.

e) Make sure to label all materials with your reference number.

f) Explain your solutions using comments in the code.

g) If you cannot complete a question, show what you attempted to do..

h) It is **not** permitted to do assignments in practical sessions unless said sessions are explicitly devoted to such purpose.

i) Assignment must be undertaken **alone**.

STUDENT ID NUMBER: ....

Part 1. The purpose of this exercise is to operate with a well-known generalisation of complex numbers, in such a way that complex and real number operations can be recovered as a special case. The **set of (Hamilton's) quaternions** is defined as the $\mathbb{R}$-vector space

$$\mathbb{H} := \{a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k} :, a, b, c, d \in \mathbb{R}\}$$

whose basis elements $\{1, \mathrm{i}, \mathrm{j}, \mathrm{k}\}$ satisfy

$$\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1. \tag{1}$$

Elements in this $\mathbb{R}$-vector space can be added entrywise

$$(a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}) + (x + y\mathrm{i} + z\mathrm{j} + t\mathrm{k}) = (a + x) + (b + y)\,\mathrm{i} + (c + z)\,\mathrm{j} + (d + t)\,\mathrm{k}, \tag{2}$$

and can be multiplied according to (1) and the distributive rule:

$$\begin{aligned}(a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}) * (x + y\mathrm{i} + z\mathrm{j} + t\mathrm{k}) &= (ax - by - cz - dt) + (ay + bx + ct - dz)\,\mathrm{i} \\ &+ (az + cx + dy - bt)\,\mathrm{j} \\ &+ (at + dx + bz - cy)\,\mathrm{k}. \end{aligned} \tag{3}$$

Two facts are immediate, and easy to check:

- Setting $b = c = d$ the above operations $+$ and $*$ in (2) and (3) are nothing but the usual real sum $+_{\mathbb{R}}$ and product $*_{\mathbb{R}}$.
- Setting $c = d$, operations $+$ and $*$ in (2) and (3) are nothing but the usual complex sum and product $+_{\mathbb{C}}$, $*_{\mathbb{C}}$.

Hence, we have the following inclusions, the two operations in each set generalizing the ones in the previous one:

$$(\mathbb{R}, +_{\mathbb{R}}, *_{\mathbb{R}}) \subseteq (\mathbb{C}, +_{\mathbb{C}}, *_{\mathbb{C}}) \subseteq (\mathbb{H}, +, *)\,,$$

You are expected to define a class `Quat` representing quaternions. Your class must use parameters, functions and operators, whether private, protected or public, for an element $q = a + b\mathrm{i} + c\mathrm{i} + d\mathrm{k} \in \mathbb{H}$, including:

- the four real entries a, b, c, d, all of them of `double` type;
- a function returning the **conjugate** $q^\star = a - b\mathrm{i} - c\mathrm{j} - d\mathrm{k}$ of q, i.e. the generalisation of complex conjugacy;
- a function returning products of the form αq, where $\alpha \in \mathbb{R}$ and $q \in \mathbb{H}$;
- a function returning the **norm** $\|q\| := \sqrt{q^\star * q} = \sqrt{q * q^\star} = \sqrt{a^2 + b^2 + c^2 + d^2}$. Needless to say, this generalises the absolute value in $\mathbb{R}$ and the modulus in $\mathbb{C}$;
- operators for equality $=$, addition $+$ and multiplication $*$, as well as functions for any simple operations derived therefrom, e.g. inversion (using (5) for example);
- specific constructors and destructors. Make sure one of the constructors is devised so as to allow you to declare a new quaternion in terms of four separate real numbers.

Your project must contain functions testing the correctness of the operators and functions in `Quat` by means of known properties. For instance:

$$q^\star = -\frac{1}{2}\left(q + \mathrm{i} * q * \mathrm{i} + \mathrm{j} * q * \mathrm{j} + \mathrm{k} * q * \mathrm{k}\right); \qquad (4)$$

$$q^{-1} = \frac{1}{\|q\|^2} q^\star; \qquad (5)$$

$$\|\alpha q\| = |\alpha|\,\|q\|, \quad \text{for every } \alpha \in \mathbb{R}; \qquad (6)$$

$$\|p * q\| = \|p\|\,\|q\|. \qquad (7)$$

Some properties can be useful to operate with vectors in $\mathbb{R}^3$. Imaginary quaternions $b\mathrm{i} + c\mathrm{i} + d\mathrm{k}$ can be used to represent three-dimensional real vectors, and some well-known vector operations can be recovered from quaternion arithmetic. If $p, q \in \mathbb{H}$ are imaginary quaternions and $\cdot$, $\times$ are the scalar and cross product in $\mathbb{R}^3$, respectively, then:

$$p \cdot q = \frac{1}{2}\left(p^\star * q + q^\star * p\right) = \frac{1}{2}\left(p * q^\star + q * p^\star\right); \qquad (8)$$

$$p \times q = \frac{1}{2}\left(p * q - q^\star * p^\star\right), \qquad (9)$$

and for every two quaternions (whether or not imaginary) $p = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}, q = x + y\mathrm{i} + z\mathrm{j} + t\mathrm{k}$, labelling $\vec{p} = b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$ and $\vec{q} = y\mathrm{i} + z\mathrm{j} + t\mathrm{k}$ their vector parts,

$$p * q = ax + a\vec{q} + x\vec{p} - \vec{p}\cdot\vec{q} + \vec{p}\times\vec{q}. \qquad (10)$$

Comments:

a) Your project must comprise separate header (`*.h`) and `.cpp` files.

b) Make sure the amount of constructors is reasonable, i.e. that you use all of them in the project.

c) The operation $*$ in $\mathbb{H}$ is in general *not* commutative: $q_1 * q_2 \neq q_2 * q_1$. Be sure to find examples for this.

d) Make sure you keep track of each property that is being checked, be it by means of exception throwing or by writing down on an output file passed by reference as an input of your test functions – *do not* declare the `ofstream` file variable as global.

e) Think of ways in which to effectively test any of (4)–(10) for finite collections of integers or rational numbers $a = \frac{m}{n}, b = \frac{p}{q}, c = \frac{r}{s}, d = \frac{t}{u}$. Test the correctness of your implementation with deliberate wrong assertions.

f) Checking (4)–(10) involves comparing `double`-type values; you can no longer assert based on *exact* equality as was the case for `int` values. Instead, you will use a predefined tolerance tol (e.g. 10^{-14}), meaning a practical threshold value for non-zero `double` outputs. For instance, we check (7) by checking $|\|pq\| - \|p\|\,\|q\|| <$ tol, e.g.

```
Quat Q1 = Quat ( 2., 1., -3., 4. ), Q2 = Quat ( -3., 4., 40., 1. );
Quat QP = Q1 * Q2;
assert ( fabs ( Q1.norm() * Q2.norm() - QP.norm() ) < tol );
```

Part 2. The purpose of this Part is to write a class representing complex numbers and operations between them. Your class should be defined in terms of the `Quat` class, using either friendship or inheritance (this is up to you):

```
class Comp {
  private: (...)
  protected: (...)
  public:
     (...)
    friend class Quat;
};
```

```
class Comp: public Quat {
  private: (...)
  protected: (...)
  public: (...)
};
```

Comments:

a) If your class `Quat` has been properly defined, the only significant role of the functions and parameters in `Comp` will be to rename and adapt a few of the existing ones in `Quat` with minimal effort. You *must not* redefine arithmetic operations for `Comp`.

b) If you are working with derived classes, bear in mind that all members and operators are automatically inherited from the base class, except for constructor(s), destructor(s), operator=`()` members and friends. Hence, you will need to redefine these for `Comp`, e.g. `Comp& Comp::operator =( const Comp& other )`, etc. In addition, you might need to add a line to the effect of `using Quat::operator=;` among the public members of `Comp`.

c) You do not need to create a special project to test this class, but are more than welcome to do so if you find enough time.

Part 3. Given a complex-valued square matrix $A \in \mathrm{Mat}_n(\mathbb{C})$, the **dominant eigenvalue** $\lambda \in \mathbb{C}$ is defined as the eigenvalue of A having the largest absolute value or modulus. You are expected to use the previous section in order to devise a program computing the dominant eigenvalue of any complex matrix, as well as an eigenvector linked to said eigenvalue.

The algorithm used to this effect will be the so-called **power iteration algorithm**. This method starts with a vector $\boldsymbol{v}_0$ (which will be accordingly changed by you if the method does not converge), and proceeds at every step by a product by A followed by normalisation:

$$\boldsymbol{v}_{k+1} := \frac{1}{\|A\boldsymbol{v}_k\|} A\boldsymbol{v}_k, \qquad k \geq 0.$$

Assuming A *does* have a dominant eigenvalue λ whose modulus is strictly larger than the rest, and that your initial condition $\boldsymbol{v}_0$ is *adequate* (in ways which we will not describe here), the sequence of vectors $\{\boldsymbol{v}_k\}_{k\geq 0}$ converges to an eigenvector $\boldsymbol{v}$ linked to the dominant eigenvalue λ: hence, $A\boldsymbol{v} = \lambda\boldsymbol{v}$.

Needless to say, you are not supposed to compute the infinite array of vectors $\{\boldsymbol{v}_k\}_{k\geq 0}$. Your method will stop whenever you have reached a step k in which two consecutive vectors of the sequence are sufficiently close to one another, i.e.

$$\|\boldsymbol{v}_k - \boldsymbol{v}_{k+1}\| < \mathrm{tol},$$

tol being a pre-set tolerance for your method. A norm difference within a fixed small tolerance will be the numerical evidence you will need for convergence. Try 10^{-14} for starters. You will need to define, *among other things*:

- a function returning the product a matrix and a vector. You can either define the function in terms of matrices whose entries are of the type `Comp`, e.g. any of

  ```
  Comp *product ( Comp **a, Comp *v, int dim );
  void product ( Comp **a, Comp *v, Comp *v2, int dim );
  ```

  etcetera, or create classes representing complex matrices and vectors (again, using friendship or inheritance if it suits you), as well as products between them.

- a function returning the norm of a complex vector. The simplest ones in this case will be the *maximum norm*:

  $$\|(v_1, \ldots, v_n)\|_\infty := \max\{|v_1|, \ldots, |v_n|\}.$$

  or the sub-2 norm generalizing the Euclidean norm:

  $$\|(v_1, \ldots, v_n)\|_2 := \sqrt{|v_1|^2 + \cdots + |v_n|^2} = \sqrt{v_1\overline{v_1} + \cdots + v_n\overline{v_n}},$$

  $\overline{z} = z^\star$ being the complex (and quaternion) conjugate of z, and $|z|$ being its modulus, expressed in Part 1 as a norm for general quaternions: $\sqrt{z * z^\star}$.

Comments:

a) Your project will include the necessary amount of header and `cpp` files.

b) Make sure not to mistake the norm of a vector with the norm of a quaternion (see Part 1); the latter, here, will be nothing but the modulus of a complex number.

c) Check whether the output $\boldsymbol{v}$ *is* an eigenvector, i.e. $\|A\boldsymbol{v} - \lambda\boldsymbol{v}\|$ is small enough.

d) Be sure to try a number of matrices, both real and complex. Each of them, along with all the relevant data, will be conveniently stored in output files. For instance, one of these files could read:

```
The matrix has entries
0. + 0. i    2.  + 0. i    3. + 0. i
1. + 0. i    20. + 0. i    4. + 0. i
0. + 0. i    1.  + 0. i    5. + 0. i

The initial condition was v0 = (1.+ 0.i, 1.+ 0.i,1.+ 0.i)
the method converged in 25 iterations using tolerance 1.e-15 to eigenvector
( 0.1077770980153898+0i, 1+0i, 0.06507003356928713+0i)
having dominant eigenvalue 20.36805723229254+0i
The norm of the difference Av-lambda v is 1.332267629550188e-15
and the method converged to eigenvector
v =  (0.700481945844.+ 0.i, 0.617096300288.+ 0.i, 1.0.+ 0.i)
linked to dominant eigenvalue lambda = 5.6170963003.+ 0.i
```