Exercise 5

*Tasks marked with a * are assessed coursework. Hand in your solutions to these via email to* `rn@ic.ac.uk`*. (Resit students submit solutions to all the tasks.) Use the subject line "*`C++ CW: surname_firstname_CW5`*", where* `surname_firstname_CW5.cpp` *is the attached file that contains your solution. The course will be assessed based on 5 pieces of coursework (25%) and an end of term driving test (75%). Your submission must be **your own work** (submissions will be checked for plagiarism), and it should compile (and run) with the GNU C++ compiler* `g++`*. The deadline for submitting this week's coursework is* **31/03/2016**.

---

1. *Mathematical vectors*
   Create a template `mathvector<>` that can be used to represent vectors in $\mathbb{K}^N$, where $\mathbb{K} = \mathbb{R}$ or $\mathbb{K} = \mathbb{C}$. For the latter you may `#include <complex>`. How you implement the vector is up to you. Define necessary and convenient member functions, and in particular appropriate constructor functions, a destructor and a function that returns the dimension of the vector. Furthermore, make sure that the following operations are properly defined for your class: `x = y`, `z = x + y`, `z = x - y`, `x = - y`, `x = r * y` and `x += y`, `x *= r`; where `x,y,z` are instances of `mathvector` and `r` is of type `T`. Finally, also overload the `operator*` function to return the scalar/dot product between two vectors, i.e. $\underline{a} \,.\, \underline{b} = \sum_{i=1}^{N} a_i \, \overline{b_i}$, where $\overline{z}$ denotes the complex conjugate of $z \in \mathbb{C}$.
   [*Hint: For older compilers the template* `T conj(const T &x) { return x; }` *may be of help.*]

2*. *Matrices*
   Create the abstract (!) base class template `mathmatrix<T>` that can be used to represent square matrices in $\mathbb{K}^{N\times N}$. Create a pure `virtual` member function to overload the `operator*` function to return the result of a matrix times `mathvector` multiplication. Provide also a pure `virtual` member function `y_eq_Ax(mathvector &y, const mathvector &x)` that computes and returns $A\,x$ in-situ on y, i.e. without creating any temporary objects.

   From this base class derive the class `fullmatrix`, which stores all elements of a matrix explicitly, and the class `diagmatrix`, which represents a diagonal matrix and hence stores only N elements. Implement the matrix times `mathvector` function for both of these classes.

   Note that you can now also derive any implicitly declared matrix from the base class, i.e. a matrix for which you ***do not*** store all N^2 elements explicitly. All you have to do, is to declare the action of that matrix on a vector.

3. *Power method*
   The largest eigenvalue (in magnitude) of a matrix $A \in \mathbb{K}^{N\times N}$ can be found using the power method:

   1. *Choose an initial guess* $\underline{x}_0 \in \mathbb{K}^N$
   2. *Let* $\underline{q} := \dfrac{\underline{x}_0}{|\underline{x}_0|}$, *where* $|\underline{x}_0|^2 = \underline{x}_0 \cdot \underline{x}_0$
   3. *For* $k = 1 : K$
   4. $\quad \underline{z} := A\,\underline{q}$
   5. $\quad \underline{q} := \dfrac{\underline{z}}{|\underline{z}|}$
   6. *end*
   7. *Return* $\lambda = \underline{q} \,.\, (A\,\underline{q})$

4*. *Power method example*
   Write a member function `mathmatrix<T>::power_method(const mathvector<T> &, int)`, which can be used to find the largest eigenvalue of e.g. the matrices $A_N \in \mathbb{R}^{N\times N}$ and $B \in \mathbb{R}^{4\times 4}$, where

$$A_N = \begin{pmatrix} 2 & -1 & 0 & \dots & 0 & -1 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & & \ddots & \ddots & & 0 \\ 0 & \dots & 0 & -1 & 2 & -1 \\ -1 & 0 & \dots & 0 & -1 & 2 \end{pmatrix} \in \mathbb{R}^{N\times N} \qquad \text{and} \qquad B = \begin{pmatrix} 3 & 1 & 2 & 5 \\ 1 & 1 & 3 & 7 \\ 2 & 3 & 2 & 4 \\ 5 & 7 & 4 & 2 \end{pmatrix} \in \mathbb{R}^{4\times 4}.$$

   Here A_N should be defined implicitly. To this end, define a separate class for it, called `CW5_matrix`.

   In particular, your code should be able to execute all of the following statements correctly.

   [*Hint: To test your code for acceptable speed, compile it with the GNU compiler with the command:* `g++ -Wall -O3 surname_firstname_CW5.cpp`.]

```
int main() {
  int N = 10000, K = 4000;
  mathvector<double> x_a;
  for (int i = 0; i < N; i++) x_a.push_back(i+1);         // x = (1,2,...,N)^T
  CW5_matrix<double> A(N);                                // matrix A
  double lambda = A.power_method(x_a, K);
  cout << "largest lambda (in modulus) for A = " << lambda << endl;
  double b[] = {1.0, 1.0, 1.0, 1.0};
  vector<double> vb(b, b+4); mathvector<double> x_b(vb);  // x = (1,1,1,1)^T
  cout << " x . x = " << x_b * x_b << endl;
  double r1[] = {3.0, 1.0, 2.0, 5.0}, r2[] = {1.0, 1.0, 3.0, 7.0},
         r3[] = {2.0, 3.0, 2.0, 4.0}, r4[] = {5.0, 7.0, 4.0, 2.0};
  vector<double> row1(r1, r1+4), row2(r2, r2+4), row3(r3, r3+4), row4(r4,r4+4);
  vector<vector<double> > BB;
  BB.push_back(row1); BB.push_back(row2); BB.push_back(row3); BB.push_back(row4);
  fullmatrix<double> B(BB);                               // matrix B
  B.y_eq_Ax(x_a, x_b); cout << x_a << endl;               // y = B * x
  lambda = B.power_method(x_b, K);
  B -= lambda;                                            // B = B - lambda * Id
  double l2 = B.power_method(x_b, K);
  B += lambda;                                            // B = B + lambda * Id
  cout << "The spectrum of B lies between " << lambda << " and " << l2 + lambda << endl;
  diagmatrix<double> D(row4);
  lambda = D.power_method(x_b, K);
  cout << "largest lambda (in modulus) for D = " << lambda << endl;
  vector<vector<complex<double> > > CC;             // ... fill CC appropriately ...
  fullmatrix<complex<double> > C(CC);               // ... etc ...
}
```

5. *Conjugate gradient solver*
For a symmetric and positive definite matrix $A \in \mathbb{R}^{N\times N}$ the system $A\,\underline{x} = \underline{b}$ can be solved using the conjugate gradient algorithm:

1. *Set* $k := 0$ *and choose an initial guess* $\underline{x}_0 \in \mathbb{R}^N$ *Let* $\underline{r}_0 := \underline{b} - A\,\underline{x}_0$, $\rho_0 := |\underline{r}_0|^2 = \underline{r}_0 \cdot \underline{r}_0$
2. *While* $k < k_{max}$ *and* $\sqrt{\rho_k} >$ *tolerance* $* \,|\underline{b}|$ *do*
3. $\quad \underline{z} := P\,\underline{r}_k, \quad \tau_k := \underline{z} \cdot \underline{r}_k$
4. $\quad$ *If* $k = 0$ *then set* $\beta := 0$ *and* $\underline{v} := \underline{0}$, *else set* $\beta := \tau_k/\tau_{k-1}$.
5. $\quad \underline{v} := \underline{z} + \beta\,\underline{v}$
6. $\quad \underline{w} := A\,\underline{v}$
7. $\quad \gamma := \tau_k/\underline{v} \cdot \underline{w}$
8. $\quad \underline{x}_{k+1} := \underline{x}_k + \gamma\,\underline{v}, \quad \underline{r}_{k+1} := \underline{r}_k - \gamma\,\underline{w}$
9. $\quad \rho_{k+1} := |\underline{r}_{k+1}|^2$
10. $\quad k := k + 1$
11. *Return* $\underline{x}_k$

Observe that the above algorithm uses only one matrix times vector multiplication per iteration (*6.*). The classical conjugate gradient method uses $P = I$ for the *preconditioner* P in *3.*, where I is the identity matrix. Otherwise P is usually a very simple matrix approximating A^{-1}.

Write a member function `mathmatrix<T>::CG_solver()` that takes the two `mathvectors` $\underline{x}_0$ and $\underline{b}$, the parameters `double` *tolerance*, `int` k_{max} and `mathmatrix` P. Let the function perform the above algorithm and return the solution in place of $\underline{x}_0$. The return value of `CG_solver()` should be the number of iterations k.

[*Hint: Although efficiency is not the priority in this exercise, try to increase the efficiency of your code, by using the method* `y_eq_Ax()` *in* 6. *and making use of the operators* `+=` *and* `*=` *throughout.*]

6. *Conjugate gradient example*
Use your solver from 5. to compute the solution of some linear systems. Include at least the example matrices A_{10000} and $\widetilde{B} := 10\,I + B$ from 4. In each case, solve the system $A\,\underline{x} = \underline{b}$ for an arbitrary right hand side $\underline{b} \neq \underline{0}$ with $\sum_i b_i = 0$. Once the solution $\underline{x}^\star$ is obtained, compute its residual $r = |A\,\underline{x}^\star - \underline{b}|$. Also compare the number of iterations when using the preconditioner $P = [\mathrm{diag}(A)]^{-1}$ compared to $P = I$.