Questions

Numerical Methods for CSE
Final Exam

Prof. Rima Alaifari

ETH Zürich, D-MATH

22 January 2018

1. *Tridiagonal system solver* (16pts)

   Important note: do **not** use built-in LSE solvers in this problem.

   Template: `1.cpp`.

   Let A be an invertible $n \times n$ matrix of real numbers. Let $a_{i,j}$ indicate its element in row i and column j. We say that A is

   - *lower bidiagonal* if $a_{i,j} = 0$ if $i \notin \{j, j+1\}$, for any i, j;
   - *upper bidiagonal* if $a_{i,j} = 0$ if $i \notin \{j-1, j\}$, for any i, j;
   - *tridiagonal* if $a_{i,j} = 0$ if $i \notin \{j-1, j, j+1\}$, for any i, j;
   - *strictly diagonally dominant* if $|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$ for any i.

$$\begin{pmatrix} d_0 & 0 & \dots & 0 & 0 \\ l_0 & d_1 & \dots & 0 & 0 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & d_{n-2} & 0 \\ 0 & 0 & \dots & l_{n-2} & d_{n-1} \end{pmatrix} \quad \begin{pmatrix} d_0 & u_0 & \dots & 0 & 0 \\ 0 & d_1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & d_{n-2} & u_{n-2} \\ 0 & 0 & \dots & 0 & d_{n-1} \end{pmatrix} \quad \begin{pmatrix} d_0 & u_0 & 0 & \dots & 0 & 0 & 0 \\ l_0 & d_1 & u_1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & l_{n-3} & d_{n-2} & u_{n-2} \\ 0 & 0 & 0 & \dots & 0 & l_{n-2} & d_{n-1} \end{pmatrix}$$

Lower bidiagonal Upper bidiagonal Tridiagonal

   (a) (2pts) Given a vector y, let x be the solution of $Ax = y$. Supposing A is *lower bidiagonal*, derive explicitly a recursive formula for x_j as a function of A, y and x_{j-1}, for every admissible j.

   Solution: From $d_0 x_0 = y_0$ we have $x_0 = y_0/d_0$. From $l_{j-1}x_{j-1} + d_j x_j = y_j$ we have $x_j = (y_j - l_{j-1}x_{j-1})/d_j$ for all $j = 1, \dots, n-1$.

   (b) (2pts) Implement a `C++/Eigen` function

   ```
   VectorXd lowBidiagSolve(const MatrixXd &A, const VectorXd &y)
   ```

   which returns the solution of `Ax = y` in $O(n)$ time, supposing `A` is *lower bidiagonal*.

   Solution:

```
VectorXd lowBidiagSolve(const MatrixXd &A, const VectorXd &y) {
    int n = y.size();
    VectorXd x(n);

    x(0) = y(0) / A(0,0);

    for (int j=1; j<n; j++) {
        x(j) = (y(j) - A(j,j-1)*x(j-1))/A(j,j);
    }

    return x;
}
```

   (c) (2pts) Given a vector y, let x be the solution of $Ax = y$. Supposing A is *upper bidiagonal*, derive explicitly a recursive formula for x_j as a function of A, y and x_{j+1}, for every admissible j.

   Solution: From $d_{n-1}x_{n-1} = y_{n-1}$ we have $x_{n-1} = y_{n-1}/d_{n-1}$. From $d_j x_j + u_{j+1}x_{j+1} = y_j$ we have $x_j = (y_j - u_{j+1}x_{j+1})/d_j$ for all $j = 0, \dots, n-2$.

   (d) (2pts) Implement a `C++/Eigen` function

`VectorXd upBidiagSolve(const MatrixXd &A, const VectorXd &y)`

which returns the solution of `Ax = y` in $O(n)$ time, supposing `A` is *upper bidiagonal*.

Solution:

```
VectorXd upBidiagSolve(const MatrixXd &A, const VectorXd &y) {
    int n = y.size();
    VectorXd x(n);

    x(n-1) = y(n-1) / A(n-1,n-1);

    for (int j=n-2; j>=0; j--) {
        x(j) = (y(j) - A(j,j+1)*x(j+1))/A(j,j);
    }

    return x;
}
```

(e) (3pts) Supposing A is *tridiagonal* and *strictly diagonally dominant*, derive on paper the LU decomposition of A.
Hint: Look for a recursive formula for the coefficients of the matrices L, U like

$$L = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ a_0 & 1 & 0 & \dots & 0 & 0 \\ 0 & a_1 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{n-2} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} b_0 & c_0 & 0 & \dots & 0 & 0 \\ 0 & b_1 & c_1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & b_{n-2} & c_{n-2} \\ 0 & 0 & 0 & \dots & 0 & b_{n-1} \end{pmatrix}. \tag{1}$$

Solution: We see immediately that it must hold $c_j = u_j$ for all j and $b_0 = d_0$. Then we can recursively obtain a_j, b_{j+1}, since from $a_j b_j = l_j$ we have $a_j = l_j / b_j$, and from $a_j c_j + b_{j+1} = d_{j+1}$ we have $b_{j+1} = d_{j+1} - a_j c_j$ for every j.

(f) (3pts) Implement a `C++/Eigen` function

`VectorXd tridiagSolve(const MatrixXd &A, const VectorXd &y)`

which returns the solution of `Ax = y` in $O(n)$ time and $O(n)$ memory, supposing `A` is *tridiagonal* and *strictly diagonally dominant*.

Solution:

```
VectorXd tridiagSolve(const MatrixXd &A, const VectorXd &y) {
    int n = y.size();
    VectorXd x(n);

    VectorXd tmp(n);
    VectorXd a(n-1);
    VectorXd b(n);
    VectorXd c(n-1);

    b(0) = A(0,0);

    for (int j=0; j<n-1; j++) {
        c(j) = A(j,j+1);
        a(j) = A(j+1,j)/b(j);
        b(j+1) = A(j+1,j+1) - a(j)*c(j);
```

```
    }

    // solve L*tmp = y
    tmp(0) = y(0);

    for (int j=1; j<n; j++) {
        tmp(j) = (y(j) - a(j-1)*tmp(j-1));
    }

    // solve U*x = tmp
    x(n-1) = tmp(n-1) / b(n-1);

    for (int j=n-2; j>=0; j--) {
        x(j) = (tmp(j) - c(j)*x(j+1))/b(j);
    }

    return x;
}
```

(g) (2pts) Is the assumption of A being *strictly diagonally dominant* necessary for the code you implemented in Point 1f to make sense (meaning that it yields a correct output)? If yes, find a matrix which is invertible and tridiagonal but **not** strictly diagonally dominant, and for which your code does **not** give the correct solution. Briefly explain why.

Solution: No it is not necessary, since the algorithm works for $A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. Notice however that for $A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$ the LU decomposition is such that $L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$, which contradicts the shape given in (1) and on which our algorithm is based. More in general, being tridiagonal and invertible does not guarantee that an LU decomposition with matrices as in (1) always exists.

2. *Sparse vectors, their convolution and FFT* (17pts)

Important note: do **not** use `Eigen` in this problem.

Template: `2.cpp`.

Given a vector `x` of `n` complex numbers, we want to represent it with a new data structure which should be particularly suitable in case most of its entries are zero.

For this purpose we implement the following two classes:

- `Pair`, which has members
  - `ind`, the index of a non-zero element of `x`;
  - `val`, the value of the non-zero element at position `ind` in `x`;

  `SparseVec`, which has members
  - `pairs`, a vector of objects `Pair` which stores the indices and values of the non-zero elements of `x`;
  - `len`, an integer which records the length of `x`.

The class `SparseVec` comes with some utility methods already implemented:

- `void append(int ind, complex val)`, which appends to `pairs` a new pair with index `ind` and value `val`;
- `void cleanup()`, which sorts `pairs` with respect to `ind` and eliminates repetitions, with the convention that if two elements have the same index their values should be added up;
- `complex get(int ind)`, which returns the element of `x` in position `ind`; works in $O(\log n)$ time and assumes that `pairs` are sorted with respect to `ind` with no repeated indices (as happens after `cleanup()` has been run);
- `SparseVec conj()`, which returns the component-wise conjugate of `x`.

Example: A representation of $x = (0, 0, 1, 0, 7 + i, 0)$ can be obtained with a `SparseVec` with `pairs` = {`Pair(ind=4,val=6-I)`, `Pair(ind=4,val=1+2I)`, `Pair(ind=2,val=1)`} and `len=6`. Running `cleanup()` on such an object, `pairs` would become {`Pair(ind=2,val=1)`, `Pair(ind=4,val=7+I)`)}.

Note: In all the subproblems below, you can assume all vectors of type `SparseVec` which are passed as inputs have sorted indices with no repetitions, i.e. `cleanup()` has been called on them.

(a) (3pts) In the class `SparseVec` implement a method

```
static SparseVec cwiseMult(const SparseVec &a, const SparseVec &b)
```

which returns the component-wise multiplication between `a` and `b` in $O(L_a + L_b)$ time, where L_a, L_b are respectively the number of non-zero elements of `a` and `b`.

Solution:

```
static SparseVec cwiseMult(const SparseVec &a, const SparseVec &b) {
    SparseVec out;
    int ia = 0;
    int ib = 0;

    while(ia < a.pairs.size() && ib < b.pairs.size()) {
        if (a.pairs[ia].ind == b.pairs[ib].ind) {
            out.append(a.pairs[ia].ind, a.pairs[ia].val * b.pairs[ib].val);
            ia++;
            ib++;
        }
        else if (a.pairs[ia].ind < b.pairs[ib].ind) {
            ia++;
        }
```

```
            else if (a.pairs[ia].ind > b.pairs[ib].ind) {
                ib++;
            }
        }

        return out;
    }
```

(b) (3pts) Implement a method

```
static SparseVec conv(const SparseVec &a, const SparseVec &b)
```

which returns the discrete convolution of `a` and `b` in $O(L_a L_b)$ time.
Warning: in this and the following subproblems do not convert the `SparseVec` to/from dense vectors.

Solution:

```
static SparseVec conv(const SparseVec &a, const SparseVec &b) {
    SparseVec out;
    for (auto x : a.pairs) {
        for (auto y : b.pairs) {
            out.append(x.ind + y.ind, x.val * y.val);
        }
    }
    return out;
}
```

(c) (5pts) Implement a method

```
static SparseVec fft(const SparseVec &x)
```

which returns the discrete Fourier transform of `x` in $O(n \log n)$ time, where n is the length of `x`. You can assume that n is a power of 2.
Hint: Adapt the divide and conquer strategy of Code 4.3.5 from the lecture notes (page 349) to `SparseVec` (do **not** use dense vectors).

Solution:

```
static SparseVec fft(const SparseVec &x) {
    int n = x.len;
    if (n<=1) return x;

    SparseVec evenElements(n/2);
    SparseVec oddElements(n/2);

    for (auto p : x.pairs) {
        if (p.ind % 2 == 0)
            evenElements.append(p.ind/2, p.val);
        else
            oddElements.append((p.ind-1)/2, p.val);
    }

    SparseVec c1 = fft(evenElements);
    SparseVec c2 = fft(oddElements);

    SparseVec out(n);
    complex omega = exp(-2.*PI/n*I);
    complex s(1.,0.);
```

```
        for (auto p : c1.pairs) {
            out.append(p.ind, p.val);
            out.append(p.ind + n/2, p.val);
        }

        for (auto p : c2.pairs) {
            out.append(p.ind, p.val * std::pow(omega, p.ind));
            out.append(p.ind + n/2, p.val * std::pow(omega, p.ind + n/2));
        }

        return out;
    }
```

The last two loops can be substituted also with the following slower loop (worth 1 point less):

```
        for (int k=0; k<n; k++) {
            tot.append(k, c1.get(k % (n/2)) + c2.get(k % (n/2))*s);
            s *= omega;
        }
```

(d) (2pts) Implement a method

```
static SparseVec ifft(const SparseVec &x)
```

which returns the inverse discrete Fourier transform of `x` in $O(n \log n)$ time. You can assume that n is a power of 2.

Hint: Recall that the inverse discrete Fourier transform of a vector `x` of length `n` can be computed as `conj(fft(conj(x)))/n`, where `conj` indicates the component-wise complex conjugate.

Solution:

```
static SparseVec ifft(const SparseVec &x) {
    SparseVec out;
    double n = x.len;
    auto tmp = fft(x.conj()).pairs;

    for (auto p : tmp) {
        complex v = std::conj(p.val)/n;
        out.append(p.ind, v);
    }
    return out;
}
```

(e) (2pts) Let `a` and `b` be two complex vectors of length n and $n+1$ respectively. Assume that n is a power of 2. Implement a method

```
static SparseVec convfft(const SparseVec &a, const SparseVec &b)
```

which returns the discrete convolution of `a` and `b` in $O(n \log n)$ time.

Solution:

```
static SparseVec convfft(SparseVec &a, SparseVec &b) {
    int N = a.len + b.len - 1;
    a.len = N;
    b.len = N;

    return ifft(cwiseMult(fft(a), fft(b)));
}
```

(f) (2pts) Can `convfft(x, y)` be much slower than `conv(x, y)` for a particular choice of `x` and `y`? Briefly motivate your answer.

Solution: Yes, for some sparse vectors `convfft(x, y)` will be much slower than `conv(x, y)`. For example, if `x` and `y` are of length n and $n + 1$ and they have all zero elements except for a 1 in position 0, their `fft` are the constant 1 vectors; then `convfft` must at least calculate all these 1's, running in $\Omega(n)$ time, while `conv` will just multiply together two numbers to get the result, running in $O(1)$ time.

3. *3-Moment system* (15pts)

Template: `3.cpp`.

Let $f_{\boldsymbol{\alpha}} : \mathbb{R} \to \mathbb{R}$ be a velocity distribution function modelled as

$$f_{\boldsymbol{\alpha}}(x) := \exp(\alpha_0 + \alpha_1 x + \alpha_2 x^2), \quad \forall x \in \mathbb{R}, \tag{2}$$

where $\boldsymbol{\alpha} \in \mathbb{R}^3$ are the model parameters. For any vector function $\mathbf{w}(x) = (w_1(x), w_2(x), w_3(x))$ we use the notation

$$\langle \mathbf{w}(x) \rangle := \left(\int_{-\infty}^{+\infty} w_1(x)dx, \int_{-\infty}^{+\infty} w_2(x)dx, \int_{-\infty}^{+\infty} w_3(x)dx \right). \tag{3}$$

Define a vector function $\boldsymbol{\phi} : \mathbb{R} \to \mathbb{R}^3$,

$$\boldsymbol{\phi}(x) := (1, x, x^2)^\top, \quad \forall x \in \mathbb{R}. \tag{4}$$

Given a vector of moments $\mathbf{u} \in \mathbb{R}^3$, the objective is to find the vector $\boldsymbol{\alpha} \in \mathbb{R}^3$ which satisfies

$$\langle \boldsymbol{\phi}(x) f_{\boldsymbol{\alpha}}(x) \rangle = \mathbf{u} \tag{5}$$

(we take for granted the existence of such an $\boldsymbol{\alpha}$).

We will use Gauss quadrature to numerically evaluate the integrals. A function '`gaussQuad`' is provided, see `gaussQuad.hpp`, to compute the Gauss nodes and weights.

(a) (2pt) Derive on paper the conversion of the improper integral $\langle \boldsymbol{\phi}(x) f_{\boldsymbol{\alpha}}(x) \rangle$ to a proper integral, that is an integral with finite upper and lower bound.
Hints: Substitute $x = \tan(s)$; $dx/ds = 1 + \tan^2(s)$.

(b) (1pt) For what values of α_2 is $\langle \boldsymbol{\phi}(x) f_{\boldsymbol{\alpha}}(x) \rangle$ finite? Give a brief reason why.

(c) (1pt) Rewrite on paper the objective (5) as a non-linear system of equations $\mathbf{F}(\boldsymbol{\alpha}) = \mathbf{0}$, specifying what $\mathbf{F}$ is.

(d) (3pts) Implement a C++/`Eigen` function for computing $\mathbf{F}(\boldsymbol{\alpha})$ from the previous task:

```
VectorXd evalF(const QuadRule& qr, const VectorXd& alpha, const VectorXd& u);
```

which uses the integral transformation from subproblem (a). A testing routine `testEvalF` has been provided to check the accuracy of your implementation.

(e) (3pts) Write explicitly on paper an iteration of the Newton method to solve the non-linear system in subproblem (c) (compute explicitly the Jacobian but not its inverse).

(f) (2pts) Implement a C++/`Eigen` function for computing the Jacobian of $\mathbf{F}(\boldsymbol{\alpha})$:

```
VectorXd evalJ(const QuadRule& qr, const VectorXd& alpha, const VectorXd& u);
```

which uses the integral transformation from subproblem (a).

(g) (2pts) Implement a C++/`Eigen` function to implement the Newton method devised in the previous step (e):

```
void newtonMethod(const QuadRule& qr, const VectorXd& u, const double atol,
        const double rtol, const int maxItr, VectorXd& alpha_);
```

Use appropriate termination criteria and the integral transformation from subproblem (a).

(h) (1pt) Test your Newton method implementation for $\mathbf{u} = (1, -1, 2)^\top$ using 40 Gauss points for integral approximation.