Signals and Systems
Lab 3

Vincent de Wit (s3038858) & Stefan Bussemaker (s2004674)

January 20, 2017

The function files that we made for this lab are shown throughout the report. The script file which was used to call the functions (Listing 11) and the utility function `powermod` (Listing 12) are shown in the Appendix. For all the exercises we tested on two test cases, the first being the test signal `a0 = [2 5 2 0 0 1 3 7]` and the second test signal `a1 = [6 9 0 14 8 1 11 13 12 2 12 13 0 12 12 11]`. Their respective transforms (in both the Fourier as the Number Theoretic domain) are defined as `f0`, `f1`, `r0` and `r1`. `asserts` are used during the exercise to test whether or not the function produced the desired output.

Exercise 1

The convolution theorem is used in this function which is DFT(a*b)=DFT(a)*DFT(b). It is implemented in the following way using the function fft: fft(a*b)=fft(a)*fft(b). `myconv(x, y)` is shown in Listing 1 and runs in O(N log N) time, since it uses `fft` and `ifft`, which both run in O($NlogN$) time.

Listing 1: *myconv.m*

```
function c = myconv(a, b)
  N = length(a);

  % Zeropadding
  a = [a zeros(1, N)];
  b = [b zeros(1, N)];

  % Convolve
  c = real(ifft(fft(a).*fft(b)));
  c = c(1:end-1);
end
```

Exercise 2

The slow DFT using a VanderMonde matrix and its reverse are defined in `slowDFT(x, y)` and `slowIFT(x, y)`. They are shown in Listing 2 and Listing 3 respectively. First, omega is determined, after which the corresponding VanderMonde matrix is created. At this point the output can be calculated, namely by multiplying the input vector with the VanderMonde matrix. The slow IFT looks a lot like the slow DFT though the VanderMonde matrix is the inverse VanderMonde matrix and the output is divided by the length of the input vector.

Listing 2: *slowDFT.m*

```
function y = slowDFT(a)
  % Input:  An n-length coefficient vector a = [a0, a1, ..., a(n-1)] and a
  %         primitive nth root of unity w, where n is a power of 2
  % Output: A vector y of values of the polynomial for a at the nth roots
  %         of unity
  N = length(a);
  w = exp(-1j*2*pi/N);                  % Omega

  V = zeros(N);                         % Make VanderMonde matrix
  for k = 0:N-1
      for n = 0:N-1
          V(k+1, n+1) = (w.^k).^n;      % F[i,j] = w^nk
      end
  end

  y = a*V;                              % y = F a
end
```

Listing 3: *slowIFT.m*

```
function a = slowIFT(y)
  % Input:  A vector y of values of the polynomial for a at the nth roots
  %         of unity and a primitive nth root of unity w, where n is a
  %         power of 2
  % Output: An n-length coefficient vector a = [a0, a1, ..., a(n-1)]
  N = length(y);
  w = exp(1j*2*pi/N);                   % Omega, without minus in exponent

  V_inv = zeros(N);                     % Make VanderMonde matrix
  for k = 0:N-1
      for n = 0:N-1
          V_inv(k+1, n+1) = (w.^k).^n;  % F[i,j] = w^-nk
      end
  end

  a = (1/N) * y * V_inv;                % a = 1/N * a F-1
end
```

Exercise 3

The slow NTT and its reverse are defined in `slowNTT(x, y)` and `slowINTT(x, y)`, which are shown in Listing 4 and Listing 5. They make use of a VanderMonde matrix and perform the NTT and the INTT. These functions look a lot like the functions in Exercise 2, though those functions used Fourier transforms and these function use Number Theoretic Transforms. This means that the VanderMonde matrix is different. To construct the VanderMonde matrix, first a prime and omega are determined based on the length of the input vector. Then the matrix consists of powers of the omega modulo the prime. Lastly the input vector is multiplied by the VanderMonde matrix to obtain the output. The slow INTT looks a lot like the slow NTT just like the functions in the previous exercise. While the VanderMonde matrix is still based on the number theoretic transform, it is inversed for the slow INTT and the ouput is multiplied with `modinvers(N,p)`. Both function make use of `powermod`, which is shown in Listing 12, which computes $a^b mod p$.

Listing 4: *slowNTT.m*

```
function y = slowNTT(a)
  % Input:  An n-length coefficient vector a = [a0, a1, ..., a(n-1)] and a
  %         primitive nth root of unity w, where n is a power of 2
  % Output: A vector y of values of the polynomial for a at the nth roots
  %         of unity

  N = length(a);
  [g, p] = rootsofunity(N);
  k = (p-1)/N;
  w = g.^k;

  V = ones(N);                              % Make Vandermonde matrix using NTT
  M = repmat(w, 1, p-1);                    % Multiplicative group modulo p
  M = mod(M.^[0:p-2], p);                   % w ^ n

  for k = 0:N-1
      for n = 0:N-1
        V(k+1, n+1) = powermod(M(k+1), n, p);   % F[i,j] = w^nk
      end
  end

  y = mod(a*V, p);                          % y = F a
end
```

Listing 5: *slowINTT.m*

```
function a = slowINTT(y)
  % Input:  A vector y of values of the polynomial for a at the nth roots
  %         of unity and a primitive nth root of unity w, where n is a
  %         power of 2
  % Output: An n-length coefficient vector a = [a0, a1, ..., a(n-1)]

  N = length(y);
  [g, p] = rootsofunity(N);
  k = (p-1)/N;
  w = g.^k;

  V_inv = ones(N);                        % Make Vandermonde matrix using NTT
  M_inv = ones(1, p-1);                   % Multiplicative group modulo p
  for l = 1:p-2;                          % w ^ -n
      M_inv(l+1) = mod(w^(p-(l+1)), p);
  end

  for k = 0:N-1
      for n = 0:N-1
        V_inv(k+1, n+1) = powermod(M_inv(k+1), n, p);    % F[i,j] = w^-nk
      end
  end

  a = mod(modinverse(N, p) * y*V_inv, p); % a = N-1 * a F-1
end
```

Exercise 4

The fast version of the DFT makes use of recursion and was invented by Cooley and Tukey. The algorithm and its reverse are coded in `fastDFT(x, y)` and `fastIFT(x, y)`, which are shown in Listing 6 and Listing 7 respectively. The fast DFT algorithm determines the Discrete Fourier Transform by smart evaluation, therefore running in $O(NlogN)$, instead of $O(N^2)$. The inverse also uses recursion and therefore also runs in $O(NlogN)$. It is almost the same as the DFT, but uses a different ω and multiplies with $1/N$ at the end.

Listing 6: *fastDFT.m*

```
function y = fastDFT(a, w)
  % Input:  An n-length coefficient vector a = [a0, a1, ..., a(n-1)] and a
  %         primitive nth root of unity w, where n is a power of 2
  % Output: A vector y of values of the polynomial for a at the nth roots
  %         of unity
  N = length(a);
  if nargin < 2                       % set w first time function is called
    w = exp(-1j*2*pi/length(a));      % act as some kind of 'wrapper'
  end                                 % is ignored within recursion

  if N == 1                           % Base case
    y = a; return
  end

  % Divide Step, which seperates even and odd indices
  x = power(w, 0);
  a_even = a(1:2:end);
  a_odd  = a(2:2:end);

  % Recursive Calls, with w^2 as (n/2)th root of unity, by the reduction
  % property
  y_even = fastDFT(a_even, power(w,2));
  y_odd  = fastDFT(a_odd, power(w,2));

  % Combine Step, using x = w^i
  y = zeros(1, N);
  for i = 1:N/2
    y(i)     = y_even(i) + x * y_odd(i);
    y(i+N/2) = y_even(i) - x * y_odd(i);
    x = x*w;
  end
end
```

Listing 7: *fastIFT.m*

```
function a = fastIFT(y, w)
  % Input:  A vector y of values of the polynomial for a at the nth roots
  %         of unity and a primitive nth root of unity w, where n is a
  %         power of 2
  % Output: An n-length coefficient vector a = [a0, a1, ..., a(n-1)]
  N = length(y);
  if nargin < 2                       % set w first time function is called
    w = exp(1j*2*pi/length(y));       % act as some kind of 'wrapper'
  end                                 % is ignored within recursion

  if N == 1                           % Base case
    a = y; return
  end

  % Divide Step, which seperates even and odd indices
  x = power(w, 0);
  a_even = y(1:2:end);
  a_odd  = y(2:2:end);

  % Recursive Calls, with w^2 as (n/2)th root of unity, by the reduction
  % property
  y_even = fastIFT(a_even, power(w,2));
  y_odd  = fastIFT(a_odd, power(w,2));

  % Combine Step, using x = w^i
  a = zeros(1, N);
  for i = 1:N/2
    a(i)     = y_even(i) + x * y_odd(i);
    a(i+N/2) = y_even(i) - x * y_odd(i);
    x = x*w;
  end

  if nargin < 2           % Perform 1/N at the end, not within the
    a = real((1/N)*a);    % recursion
  end
end
```

Exercise 5

The recursive version using NTT is `fastNTT(x, y)` and is shown in Listing 8. Its inverse is `fastINTT(x, y)` and is shown in Listing 9. In the previous exercise the recursive algorithm was used to do a Discrete Fourier Transform, therefore using a an omega which was an e-power. In this version of the algorithm the Number Theoretic Transform is used like in Exercise 2. These programs use the same structure as the regular FFT, but now use a primitive Nth root of unity in the multiplicative group modulo prime.

Listing 8: *fastNTT.m*

```
function y = fastNTT(a, w, p)
  % Input:  An n-length coefficient vector a = [a0, a1, ..., a(n-1)] and a
  %         primitive nth root of unity w, where n is a power of 2
  % Output: A vector y of values of the polynomial for a at the nth roots
  %         of unity
  N = length(a);
  if nargin < 2                 % set w first time function is called
    [g, p] = rootsofunity(N);   % it is already set in the recursion step
    k = (p-1)/N;
    w = g^k;
  end

  if N == 1                     % Base case
    y = a; return
  end

  % Divide Step, which seperates even and odd indices
  x = power(w, 0);
  a_even = a(1:2:end);
  a_odd  = a(2:2:end);

  % Recursive Calls, with w^2 as (n/2)th root of unity, by the reduction
  % property
  y_even = fastNTT(a_even, rem(power(w,2), p), p);
  y_odd  = fastNTT(a_odd, rem(power(w,2), p), p);

  % Combine Step, using x = w^i
  y = zeros(1, N);
  for i = 1:N/2
    y(i)     = rem(y_even(i) + rem(x * y_odd(i), p), p);
    y(i+N/2) = rem(y_even(i) - rem(x * y_odd(i), p) + p, p);
    x = rem(x*w, p);
  end
end
```

Listing 9: *fastINTT.m*

```
function a = fastINTT(y, w, p)
  % Input:  A vector y of values of the polynomial for a at the nth roots
  %         of unity and a primitive nth root of unity w, where n is a
  %         power of 2
  % Output: An n-length coefficient vector a = [a0, a1, ..., a(n-1)]
  N = length(y);
  if nargin < 2                 % set w first time function is called
    [g, p] = rootsofunity(N);   % it is already set in the recursion step
    k = (p-1)/N;
    w = g^k;
  end

  if N == 1                     % Base case
    a = y; return
  end

  % Divide Step, which seperates even and odd indices
  x = power(w, 0);
  a_even = y(1:2:end);
  a_odd  = y(2:2:end);

  % Recursive Calls, with w^2 as (n/2)th root of unity, by the reduction
  % property
  y_even = fastINTT(a_even, rem(power(w,2), p), p);
  y_odd  = fastINTT(a_odd, rem(power(w,2), p), p);

  % Combine Step, using x = w^i
  a = zeros(1, N);
  for i = 1:N/2
    a(i)     = rem(y_even(i) + rem(x * y_odd(i), p), p);
    a(i+N/2) = rem(y_even(i) - rem(x * y_odd(i), p) + p, p);
    x = rem(x*w, p);
  end

  if nargin < 2                       % Perform N-1 at final step, not
    a = rem(modinverse(N, p) * a, p); % within the recursion
    a(2:end) = a(end:-1:2);
  end
end
```

Exercise 6

The adapted version of `myconv` using NTT is `myconv_ntt(x, y)` and is shown in Listing 10. It runs in O($NlogN$) time, since it uses `fastNTT` and `fastINTT`, which both run in O($NlogN$) time. We noticed that the function `myconv_ntt` does not always result in the correct output. After thinking about the reason behind this and trying out test cases, we figured the problem lies within the functions `fastNTT` and `fastINTT`. The problem arises because the result of the NTT functions is always in modulo prime. Therefore we started thinking about the Number Theoretic Transform. We think that using the NTT limits the ability of the algorithm to represent signals which contain element whos values are larger than the used prime. This would explain the faults in `myconv_ntt`.

Listing 10: *myconv_ntt.m*

```
function c = myconv_ntt(a, b)
  N = length(a);

  % Zeropadding
  a = [a zeros(1, N)];
  b = [b zeros(1, N)];

  x = fastNTT(a);
  y = fastNTT(b);
  z = x.*y;

  c = fastINTT(z);
  c = c(1:end-1);
end
```

Appendix

Listing 11: *Script file for lab 3*

```
% Test cases
a0 = [2 5 2  0  0 1  3 7];        % Signal 0 of length 2^3, given by TAs
r0 = [3 5 1 12 11 7 10 1];        % NTT coefficients of a0, for testing
f0 = fft(a0);                     % FFT coefficients of a0, for testing

a1 = [6 9  0 14  8 1 11 13 12  2 12 13 0 12 12 11];
r1 = [0 7 13  1 14 9 14  6  3 11 10 13 2  8  3 16];
f1 = fft(a1);

b0 = randi([0 10],1,length(a0));  % Random signal, same length as a0
b1 = randi([0 10],1,length(a1));  % Random signal, same length as a1

tol = eps(1e10); % Tolerance for equality testing of floats, from:
                 % https://mathworks.com/help/matlab/ref/eq.html#bt2klek-3

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% %%
% Make a function myconv(x,y) that returns the convolution of the signals
% x and y

c1 = conv(a0, b0);    c2 = myconv(a0, b0);
c3 = conv(a1, b1);    c4 = myconv(a1, b1);

assert(all(abs(c1-c2) < tol), 'Ex1: myconv fails for a0, b0');
assert(all(abs(c3-c4) < tol), 'Ex1: myconv fails for a0, b0');


%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% %%
% Make a (very slow) DFT by constructing a vanderMonde matrix, and
% multiplying the input with this matrix. Compare your result with the
% built-in function FFT. Make also the inverse IFT

y0 = slowDFT(a0);       c0 = slowIFT(y0);
y1 = slowDFT(a1);       c1 = slowIFT(y1);

assert(all(abs(y0-f0) < tol), 'Ex2: slowDFT fails for a0');
assert(all(abs(c0-a0) < tol), 'Ex2: slowDFT+slowIFT fail for a0');
assert(all(abs(y1-f1) < tol), 'Ex2: slowDFT fails for a1');
assert(all(abs(c1-a1) < tol), 'Ex2: slowDFT+slowIFT fail for a1');


%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% %%
% Repeat the previous exercise using prime numbers and roots of unity.

y0 = slowNTT(a0);   c0 = slowINTT(y0);
y1 = slowNTT(a1);   c1 = slowINTT(y1);

assert(isequal(y0, r0), 'Ex3: slowNTT fails for a0');
assert(isequal(c0, a0), 'Ex3: slowNTT+slowINTT fail for a0');
assert(isequal(y1, r1), 'Ex3: slowNTT fails for a1');
assert(isequal(c1, a1), 'Ex3: slowNTT+slowINTT fail for a1');
```

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% %%
% Implement the FFT and IFFT yourself (use a recursive impementation: see
% slide 36 of lecture 5). Compare with the built-in FFT.

y0 = fastDFT(a0);       c0 = fastIFT(y0);
y1 = fastDFT(a1);       c1 = fastIFT(y1);

assert(all(abs(y0-f0) < tol), 'Ex4: fastDFT fails for a0');
assert(all(abs(c0-a0) < tol), 'Ex4: fastDFT+fastIFT fail for a0');
assert(all(abs(y1-f1) < tol), 'Ex4: fastDFT fails for a1');
assert(all(abs(c1-a1) < tol), 'Ex4: fastDFT+fastIFT fail for a1');


%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% %%
% Modify the code of the previous exercise to implement the NTT and its
% inverse.

y0 = fastNTT(a0);   c0 = fastINTT(y0);
y1 = fastNTT(a1);   c1 = fastINTT(y1);

assert(isequal(y0, r0), 'Ex5: fastNTT fails for a0');
assert(isequal(c0, a0), 'Ex5: fastNTT+fastINTT fail for a0');
assert(isequal(y1, r1), 'Ex5: fastNTT fails for a1');
assert(isequal(c1, a1), 'Ex5: fastNTT+fastINTT fail for a1');


%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% %%
% Repeat the first exercise (myconv), but now using NTTs.

c1 = conv(a0, b0);    c2 = myconv_ntt(a0, b0);  % Answer still in mod p!
c3 = conv(a1, b1);    c4 = myconv_ntt(a1, b1);  % Answer still in mod p!

assert(all(abs(c1-c2) < tol), 'Ex6: myconv_ntt fails for a0, b0');
assert(all(abs(c3-c4) < tol), 'Ex6: myconv_ntt fails for a1, b1');
```

Listing 12: *Function file for powermod*

```
function res = powermod(a, b, p)
  % Powermod is a utility function to compute a^b mod p
  % This function is introduced to overcome Matlabs inaccuracy when dealing
  % with powers above 10e16, see:
  % http://mathworks.com/matlabcentral/newsreader/view_thread/7898
  res = 1;
  for i = 1:b
      res = mod(res*a, p);
  end
end
```