POLAR **Successive Cancellation** MATLAB Codes and Functions

```
% assumed reliability sequence
Q =[0 1 2 4 8 16 32 3 5 64 9 6 17 10 18 128 12 33 65 20 256 34 24 36 7 129
66 512 11 40 68 130 19 13 48 14 72 257 21 132 35 258 26 513 80 37 25 22 136
260 264 38 514 96 67 41 144 28 69 42 516 49 74 272 160 520 288 528 192 544
70 44 131 81 50 73 15 320 133 52 23 134 384 76 137 82 56 27 97 39 259 84 138
145 261 29 43 98 515 88 140 30 146 71 262 265 161 576 45 100 640 51 148 46
75 266 273 517 104 162 53 193 152 77 164 768 268 274 518 54 83 57 521 112
135 78 289 194 85 276 522 58 168 139 99 86 60 280 89 290 529 524 196 141
101 147 176 142 530 321 31 200 90 545 292 322 532 263 149 102 105 304 296
163 92 47 267 385 546 324 208 386 150 153 165 106 55 328 536 577 548 113 154
79 269 108 578 224 166 519 552 195 270 641 523 275 580 291 59 169 560 114
277 156 87 197 116 170 61 531 525 642 281 278 526 177 293 388 91 584 769 198
172 120 201 336 62 282 143 103 178 294 93 644 202 592 323 392 297 770 107
180 151 209 284 648 94 204 298 400 608 352 325 533 155 210 305 547 300 109
184 534 537 115 167 225 326 306 772 157 656 329 110 117 212 171 776 330 226
549 538 387 308 216 416 271 279 158 337 550 672 118 332 579 540 389 173 121
553 199 784 179 228 338 312 704 390 174 554 581 393 283 122 448 353 561 203
                                                          85 299 354
211 401 185 396 344 586 645 593 535 240 206 95 327 564 800 402 356 307 301
417 213 568 832 588 186 646 404 227 896 594 418 302 649 771 360 539 111 331
214 309 188 449 217 408 609 596 551 650 229 159 420 310 541 773 610 657 333
119 600 339 218 368 652 230 391 313 450 542 334 233 555 774 175 123 658 612
341 777 220 314 424 395 673 583 355 287 183 234 125 557 660 616 342 316 241
778 563 345 452 397 403 207 674 558 785 432 357 187 236 664 624 587 780 705
126 242 565 398 346 456 358 405 303 569 244 595 189 566 676 361 706 589 215
786 647 348 419 406 464 680 801 362 590 409 570 788 597 572 219 311 708 598
601 651 421 792 802 611 602 410 231 688 653 248 369 190 364 654 659 335 480
315 221 370 613 422 425 451 614 543 235 412 343 372 775 317 222 426 453 237
559 833 804 712 834 661 808 779 617 604 433 720 816 836 347 897 243 662 454
318 675 618 898 781 376 428 665 736 567 840 625 238 359 457 399 787 591 678
434 677 349 245 458 666 620 363 127 191 782 407 436 626 571 465 681 246 707
350 599 668 790 460 249 682 573 411 803 789 709 365 440 628 689 374 423 466
793 250 371 481 574 413 603 366 468 655 900 805 615 684 710 429 794 252 373
605 848 690 713 632 482 806 427 904 414 223 663 692 835 619 472 455 796 809
714 721 837 716 864 810 606 912 722 696 377 435 817 319 621 812 484 430 838
667 488 239 378 459 622 627 437 380 818 461 496 669 679 724 841 629 351 467
438 737 251 462 442 441 469 247 683 842 738 899 670 783 849 820 728 928 791
367 901 630 685 844 633 711 253 691 824 902 686 740 850 375 444 470 483 415
485 905 795 473 634 744 852 960 865 693 797 906 715 807 474 636 694 254 717
575 913 798 811 379 697 431 607 489 866 723 486 908 718 813 476 856 839 725
698 914 752 868 819 814 439 929 490 623 671 739 916 463 843 381 497 930 821
726 961 872 492 631 729 700 443 741 845 920 382 822 851 730 498 880 742 445
471 635 932 687 903 825 500 846 745 826 732 446 962 936 475 853 867 637 907
487 695 746 828 753 854 857 504 799 255 964 909 719 477 915 638 748 944 869
491 699 754 858 478 968 383 910 815 976 870 917 727 493 873 701 931 756 860
499 731 823 922 874 918 502 933 743 760 881 494 702 921 501 876 847 992 447
733 827 934 882 937 963 747 505 855 924 734 829 965 938 884 506 749 945 966
755 859 940 830 911 871 639 888 479 946 750 969 508 861 757 970 919 875 862
```

```
758 948 977 923 972 761 877 952 495 703 935 978 883 762 503 925 878 735 993
885 939 994 980 926 764 941 967 886 831 947 507 889 984 751 942 996 971 890
509 949 973 1000 892 950 863 759 1008 510 979 953 763 974 954 879 981  982
927 995 765 956 887 985 997 986 943 891 998 766 511 988 1001 951 1002  893
975 894 1009 955 1004 1010 957 983 958 987 1012 999 1016 767 989 1003  990
1005 959 1011 1013 895 1006 1014 1017 1018 991 1020 1007 1015 1019 1021 1022
1023]+1;
%openExample('comm/QFunctionResultsAndPlotExample')
% initialization of given terms
```

Taking Inputs from User

```
N = input('Enter the value of N: ');
if(N>1024)
    disp('Please enter the value of N between 2 and 1024');
end
K = input('Enter the value of K:')
n=log2(N);
l=1;    % index pointer for determining elgible bit indexes
        % from reliability sequence between 1 and N

Q1=[];  % reliability sequence for N bits
for i=1:N
    while Q(l)>N
         l=l+1;
    end
    Q1(i)=Q(l);
    l=l+1;
end



%% Monte-Carlo Simulation
Nsim=100;
%Nbiterrs=0;
Nbiterrs_array =zeros(1, length(0.5:0.5:10));
Nbituncoded_array =zeros(1, length(0.5:0.5:10));
%Nblkerrs=0;
index = 1;
arr=zeros(1, length(0.5:0.5:10));
Pc = zeros(20,imax);
count=0;
```

Performing SNR for 0.5 to 10

```
for SNR_dB=0.5:0.5:10
    Nbiterrs=0;
```

Performing Monte- Carlo for each SNR

```
    for i=1:Nsim

        %Varying the SNR values from 0.5 to 10 dB

        msg=randi([0,1],1,K);    % K bits message vector

        u = [];   % initialize input sequence

        for i=1:N-K    % first N-K bits are the frozen bits
            u(Q1(i))=0;
        end

        t=1; % index for msg bits
        for i=N-K+1:N   % next K bits are message bits
            u(Q1(i))=msg(t);
            t=t+1;
        end

        input_seq=[];    % input sequence to be used later to verify decoded
sequence
        for i=1:N
            input_seq(i)=u(i);
        end


        % Now find the codeword c=uG where u=input sequence and G=polar
matrix
        % Initialize parameters
```

Encoding process

```
        depth = log2(N) - 1;
        % number of bits that to be module sum at the dth depth
        total_at_that_depth = 1;

        % loop is going to n-1 to 0 in down_ward to up_ward
            while depth >=0
                %inisalizing temp variable at index one
                temp = 1 ;
                % going 1 to N for all depth levels
                while temp<=N
                % storing first_bit from that index to  total_at_that_depth
-1 ;

                first_msg_bit = u(temp : temp + total_at_that_depth -1 );
```

```
                % storing second_bit from that index + total_at_that_depth
to  temp + 2 *total_at_that_depth-1;

                second_msg_bit = u(temp + total_at_that_depth : temp + 2
*total_at_that_depth-1 );

                % storing message
                    u(temp :  temp + 2* total_at_that_depth -1 ) =
[mod(first_msg_bit + second_msg_bit, 2 ) second_msg_bit] ;

                    %updating the index as exponantially as graph's level
nodes
                    %increasing exponentially
                   temp = temp + 2 * total_at_that_depth ;
                end

                total_at_that_depth = 2 * total_at_that_depth ;
                depth  = depth - 1 ;
            end

        %codeword is c
            c=u;
            y=[]; % output vector after BPSK mapping
            for i=1:N
                   if(c(i)==0)
                        y(i) = 1;
                   else
                        y(i) = -1;
                   end
            end
            SNR_linear = 10 ^ (SNR_dB / 10);
            sigma = sqrt(1 /SNR_linear);

            r = y + sigma*randn(size(y));  %% r is the received codeword
after passing through AWGN
            x=[];
            for i=1:N
                if(i<=N-K)
                    x(Q1(i))=1 ;
                else
                    x(Q1(i))=0 ;
                end
            end
```

Decoding Process

```
            [res, q]=polar_decode(r,x);      %% We store the decoded
sequence in the 'res' vector                    %% After polar decoding ,
we print the obtained sequence
```

```
        % msg_receive=res(n+1,Q1(N-K+1:end));
        %% bit errors
        disp(res)
        %Nerrs = sum(input_seq ~=res);
        %err_cnt = N
        for i = 1:N
            if res(i)~=input_seq(i)
                flag=0;
                break;
            else
                flag=1;
            end
        end
        count=count+flag;
    end

   % BER_sim = (Nbiterrs/N/Nsim);
   % BER_uncoded=qfunc(sqrt(2*SNR_dB));
  %  Nbituncoded_array(index) =BER_uncoded;
      % Increment index for the next SNR value
    arr(index)=count/Nsim;
    index = index + 1;
end


 % FER_sim = Nblkerrs/Nsim;
  %disp([SNR_dB FER_sim BER_sim Nblkerrs Nbiterrs Nsim])
SNR_values=0.5:0.5:10;
plot(SNR_values,arr)
%plot(SNR_values,y1_axis,'r-');
```

Ploting Graphs

```
%yscale log
xlabel('Iterations (SNR_Values)');
ylabel('Pc(i) Probability');
title('Probability of successful decoding');



%% We observed that as the value of SNR increase from 0.5 to 10 , the error
in the decoding reduces
%% significantly
```

Decoding process for SC

```
function [u,v] = polar_decode(y, froz_bit)

%% This is the recursive function for successive cancellation polar decoding

%% y is the belief vector i.e the codeword vector when passed through BPSK
mapping
%% and AWGN channel

N = length(y);  %% no. of bits of sequence y

if N == 2   %% base condition

    L1 = f(y(1), y(2));     %% f is the function for left sided nodes
                            %% SPC coding
    u = zeros(1, 2);
    if froz_bit(1) == 1     %% the estimate for u1 is 0
        u(1) = 0;
    elseif L1 >= 0
        u(1) = 0;
    else
        u(1) = 1;
    end

    L2 = g(y(1), y(2), u(1));     %% g is the function of right sided nodes
                                  %% repetition coding
    if froz_bit(2) == 1
        u(2) = 0;
    elseif L2 >= 0
        u(2) = 0;
    else
        u(2) = 1;
    end

    v = zeros(1, 2);
    v(1) = bitxor(u(1), u(2));
    v(2) = u(2);

else

    Traverse_left = zeros(1, N/2);
    for index = 1:(N/2)
        Traverse_left(index) = f(y(index), y(index+N/2));
    end

    f_ind1 = froz_bit(1:(N/2));

    [u0, v0] = polar_decode(Traverse_left, f_ind1);
```

```
    Traverse_right = zeros(1, N/2);
    for index = 1:(N/2)
        Traverse_right(index) = g(y(index), y(index+N/2), v0(index));
    end

    f_ind2 = froz_bit((N/2 +1):N);

    [u1, v1] = polar_decode(Traverse_right, f_ind2);

    u = [u0, u1];

    v = zeros(1, N);
    for index = 1:(N/2)
        v(index) = bitxor(v0(index), v1(index));
        v(index+N/2) = v1(index);
    end

end

end
```

G function for SC

```
function result = g(x, y, u)
if u == 0
  result = x + y;
elseif u == 1
  result = y - x;
  end
end
```

F function for SC Soft decisions

```
function result = f(x, y)
result = sign(x) * sign(y) * min(abs(x), abs(y));
end

%% Min sum algorithm for soft decisions
```

Polar Successive_Cancellation Simulation

```
% Loading Channel indices in increasing order of reliability
Q1=[0 1 2 4 8 16 32 3 5 64 9 6 17 10 18 128 12 33 65 20 256 34 24 36 7 129
66 512 11 40 68 130 ...
    19 13 48 14 72 257 21 132 35 258 26 513 80 37 25 22 136 260 264 38 514
96 67 41 144 28 69 42 ...
    516 49 74 272 160 520 288 528 192 544 70 44 131 81 50 73 15 320 133 52
23 134 384 76 137 82 56 27 ...
    97 39 259 84 138 145 261 29 43 98 515 88 140 30 146 71 262 265 161 576
45 100 640 51 148 46 75 266 273 517 104 162 ...
    53 193 152 77 164 768 268 274 518 54 83 57 521 112 135 78 289 194 85 276
522 58 168 139 99 86 60 280 89 290 529 524 ...
    196 141 101 147 176 142 530 321 31 200 90 545 292 322 532 263 149 102
105 304 296 163 92 47 267 385 546 324 208 386 150 153 ...
    165 106 55 328 536 577 548 113 154 79 269 108 578 224 166 519 552 195
270 641 523 275 580 291 59 169 560 114 277 156 87 197 ...
    116 170 61 531 525 642 281 278 526 177 293 388 91 584 769 198 172 120
201 336 62 282 143 103 178 294 93 644 202 592 323 392 ...
    297 770 107 180 151 209 284 648 94 204 298 400 608 352 325 533 155 210
305 547 300 109 184 534 537 115 167 225 326 306 772 157 ...
    656 329 110 117 212 171 776 330 226 549 538 387 308 216 416 271 279 158
337 550 672 118 332 579 540 389 173 121 553 199 784 179 ...
    228 338 312 704 390 174 554 581 393 283 122 448 353 561 203 63 340 394
527 582 556 181 295 285 232 124 205 182 643 562 286 585 ...
    299 354 211 401 185 396 344 586 645 593 535 240 206 95 327 564 800 402
356 307 301 417 213 568 832 588 186 646 404 227 896 594 ...
    418 302 649 771 360 539 111 331 214 309 188 449 217 408 609 596 551 650
229 159 420 310 541 773 610 657 333 119 600 339 218 368 ...
    652 230 391 313 450 542 334 233 555 774 175 123 658 612 341 777 220 314
424 395 673 583 355 287 183 234 125 557 660 616 342 316 ...
    241 778 563 345 452 397 403 207 674 558 785 432 357 187 236 664 624 587
780 705 126 242 565 398 346 456 358 405 303 569 244 595 ...
    189 566 676 361 706 589 215 786 647 348 419 406 464 680 801 362 590 409
570 788 597 572 219 311 708 598 601 651 421 792 802 611 ...
    602 410 231 688 653 248 369 190 364 654 659 335 480 315 221 370 613 422
425 451 614 543 235 412 343 372 775 317 222 426 453 237 ...
    559 833 804 712 834 661 808 779 617 604 433 720 816 836 347 897 243 662
454 318 675 618 898 781 376 428 665 736 567 840 625 238 ...
    359 457 399 787 591 678 434 677 349 245 458 666 620 363 127 191 782 407
436 626 571 465 681 246 707 350 599 668 790 460 249 682 ...
    573 411 803 789 709 365 440 628 689 374 423 466 793 250 371 481 574 413
603 366 468 655 900 805 615 684 710 429 794 252 373 605 ...
    848 690 713 632 482 806 427 904 414 223 663 692 835 619 472 455 796 809
714 721 837 716 864 810 606 912 722 696 377 435 817 319 ...
    621 812 484 430 838 667 488 239 378 459 622 627 437 380 818 461 496 669
679 724 841 629 351 467 438 737 251 462 442 441 469 247 ...
    683 842 738 899 670 783 849 820 728 928 791 367 901 630 685 844 633 711
253 691 824 902 686 740 850 375 444 470 483 415 485 905 ...
```

```
    795 473 634 744 852 960 865 693 797 906 715 807 474 636 694 254 717 575
913 798 811 379 697 431 607 489 866 723 486 908 718 813 ...
    476 856 839 725 698 914 752 868 819 814 439 929 490 623 671 739 916 463
843 381 497 930 821 726 961 872 492 631 729 700 443 741 ...
    845 920 382 822 851 730 498 880 742 445 471 635 932 687 903 825 500 846
745 826 732 446 962 936 475 853 867 637 907 487 695 746 ...
    828 753 854 857 504 799 255 964 909 719 477 915 638 748 944 869 491 699
754 858 478 968 383 910 815 976 870 917 727 493 873 701 ...
    931 756 860 499 731 823 922 874 918 502 933 743 760 881 494 702 921 501
876 847 992 447 733 827 934 882 937 963 747 505 855 924 ...
    734 829 965 938 884 506 749 945 966 755 859 940 830 911 871 639 888 479
946 750 969 508 861 757 970 919 875 862 758 948 977 923 ...
    972 761 877 952 495 703 935 978 883 762 503 925 878 735 993 885 939 994
980 926 764 941 967 886 831 947 507 889 984 751 942 996 ...
    971 890 509 949 973 1000 892 950 863 759 1008 510 979 953 763 974 954
879 981 982 927 995 765 956 887 985 997 986 943 891 998 766 ...
    511 988 1001 951 1002 893 975 894 1009 955 1004 1010 957 983 958 987
1012 999 1016 767 989 1003 990 1005 959 1011 1013 895 1006 1014 1017 1018 ...
    991 1020 1007 1015 1019 1021 1022 1023]+1;

%We are giving pre define length of codeword
N = 1024;

% List size which is store how many out put at a time you have to take
nL = 4;

% Length of CRC
% Bassically , CRC is used for check your output is correct or not and it is
based on polynomial
crc_len = 16;

% Select the CRC polynomial

    % CRC Generator
    crcGen = comm.CRCGenerator([16 15 2 0]);

    % CRC Detector
    crcDet = comm.CRCDetector([16 15 2 0]);


% Extracting channel indices from Q for smaller N
Q = Q(Q<=N);



% Length of message vector
K = 512;

% Number of blocks transmitted
Nsim = 10;
```

```
% No. of samples in EBN0 range
nSam = 10;

% Eb/N0 range of signal in dB
EbN0dB = 1:1:10;


% Vector to store Bit Error Rate for theoratical
Theoratical = zeros(1,length(EbN0dB));

% Vector to store Bit Error Rate for simulation
simulation = zeros(1,length(EbN0dB));

% Indices corresponding to data channels
msg_pos = Q(N-K+1:end);

% Boolean array with information nodes pos = 1
msg_bit_array = zeros(1,N);
msg_bit_array(msg_pos) = 1;

index=1 ;
% Simulation over the range of EbN0
```

Performing SCL for SNR range 1 to 10

```
for SNR_dB = 1:1:10

    % Display the simulation status
    disp("counter " + SNR_dB + " of " + length(EbN0dB));
    disp(' ');

    %Initialising the number of errors to zero for each sigma


    %Bit_Error variable
    err_SCL = 0;
    %Frame_Error variable
    fer_SCL = 0;

    Nsim_count = 0;
    % Simulation over the message vectors
```

Monte- Carlo Simulation

```
    parfor i_m = 1:Nsim

        Nsim_count = Nsim_count + 1;
        % Generating a random message vector
        msg = randi([0,1],1,K - crc_len);
```

```
        msg = transpose(crcGen(transpose(msg)));


        u = [];   % initialize input sequence

        for i=1:N-K    % first N-K bits are the frozen bits
            u(Q1(i))=0;
        end

        t=1; % index for msg bits
        for i=N-K+1:N   % next K bits are message bits
            u(Q1(i))=msg(t);
            t=t+1;
        end

        input_seq=[];    % input sequence to be used later to verify decoded
sequence
        for i=1:N
            input_seq(i)=u(i);
        end


        % Initialize parameters
```

Performing Encoding for Msg length K

```
        depth = log2(N) - 1;
        % number of bits that to be module sum at the dth depth
        total_at_that_depth = 1;

        % loop is going to n-1 to 0 in down_ward to up_ward
            while depth >=0
                %inisalizing temp variable at index one
                temp = 1 ;
                % going 1 to N for all depth levels
                while temp<=N
                % storing first_bit from that index to  total_at_that_depth
-1 ;

                first_msg_bit = u(temp : temp + total_at_that_depth -1 );

                 % storing second_bit from that index + total_at_that_depth
to  temp + 2 *total_at_that_depth-1;

                second_msg_bit = u(temp + total_at_that_depth : temp + 2
*total_at_that_depth-1 );

                % storing message
```

```
                    u(temp :  temp + 2* total_at_that_depth -1 ) =
[mod(first_msg_bit + second_msg_bit, 2 ) second_msg_bit] ;

                    %updating the index as exponantially as graphs level
nodes
                    %increasing exponentially
                   temp = temp + 2 * total_at_that_depth ;
                end

                total_at_that_depth = 2 * total_at_that_depth ;
                depth  = depth - 1 ;
            end

           %codeword is c
            cword=u;
              y=[];
```

Performing BPSK

```
            for i=1:N
                  if(cword(i)==0)
                       y(i) = 1;
                  else
                       y(i) = -1;
                  end
            end
```

Performing AWGN

```
            SNR_linear = 10 ^ (SNR_dB / 10);
            sigma = sqrt(1  /SNR_linear);
            LLR = y + sigma*randn(size(y));
```

Decoding the received message

```
        [msg_hat,PM] = decode_SCL(LLR, N, nL, msg_bit_array, msg_pos);

        crc_bit_error = zeros(1,nL);
        for i_crc = 1:nL
            [~, crc_bit_error(i_crc)] = crcDet(transpose(msg_hat(i_crc,:)));
        end

        PM(crc_bit_error > 0) = inf;
        [~, min_ind] = min(PM);
        msg_hat = msg_hat(min_ind,:);
```

Checking for the Errors

```
        nberr = sum(xor(msg(1:K),msg_hat(1:K)));
        err_SCL = err_SCL + nberr;
        if (nberr > 0)
            fer_SCL = fer_SCL + 1;
        end

    end
    % Bit Error Rate & Frame Error Rate
    Theoratical(1,SNR_dB) = err_SCL/(Nsim_count*(K-crc_len));
    simulation(1,SNR_dB) = fer_SCL/(Nsim_count);

end
```

```
counter 1 of 10

counter 2 of 10

counter 3 of 10

counter 4 of 10

counter 5 of 10

counter 6 of 10

counter 7 of 10

counter 8 of 10

counter 9 of 10

counter 10 of 10
```

```
time = toc;
disp("Time taken for  simulations is " + time + " secs");
```

```
Time taken for  simulations is 523703.8553 secs
```

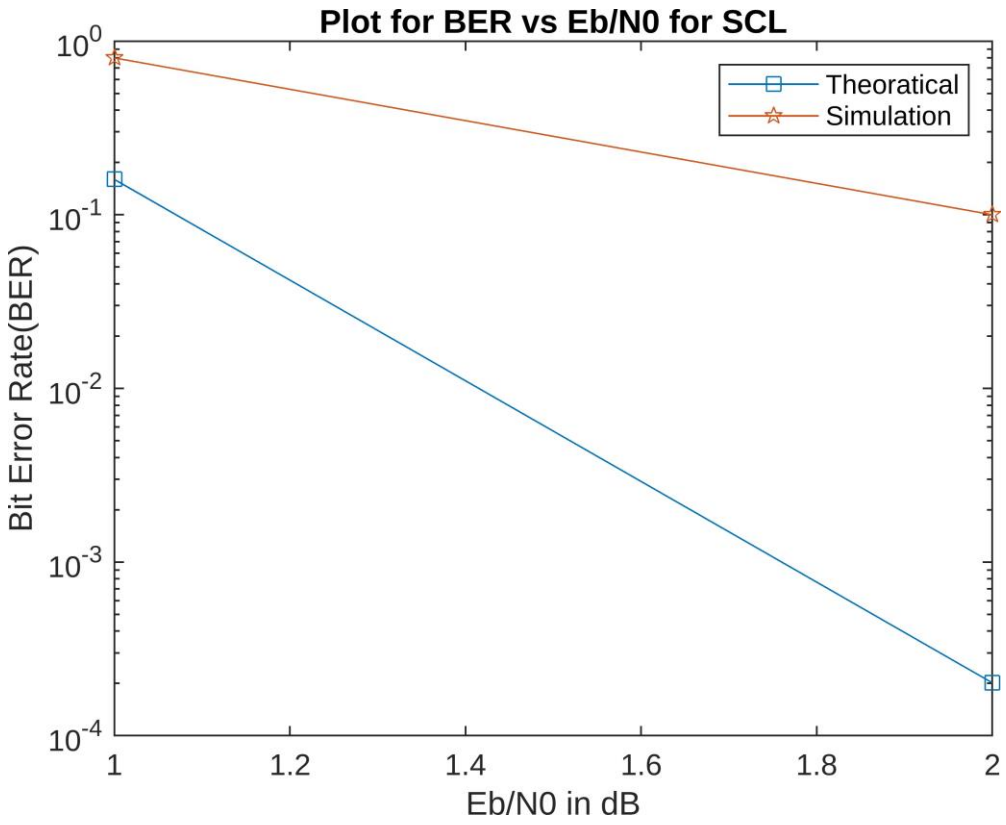
```
%% plot for Theoratical Vs Simulation
```

Ploting the Graphs

```
figure(1)
semilogy(EbN0dB,Theoratical,'-s');
hold on;
semilogy(EbN0dB,simulation,'-p');

legend('Theoratical','Simulation');
str = sprintf('Plot for BER vs Eb/N0 for SCL');
title(str);
xlabel('Eb/N0 in dB');
```

```
ylabel('Bit Error Rate(BER)');
```



SCL_DECODING FUNCTION

```
function [msg_hat,PM] = decode_SCL(LLR, N, l, info_check_vec, data_pos)
```

```
    % f function
    f_minsum = @(a,b) (1-2*(a<0)).*(1-2*(b<0)).*min(abs(a),abs(b));
```

```
    % beliefs
    L = zeros(l,n+1,N);

    % decisions
    ucap = zeros(l,n+1,N)-2;

    % index ordering
    ind_ord_mat = zeros(l,n+1,N);

    % vector to check status of node -- left or right or parent propagation
    ns = zeros(1,2*N-1);
```

F (MinSum) Function

G Function

```
    % g function
    g_minsum = @(a,b,c) b+(1-2*c).*a;

    % belief initialisation
    for i = 1:l
        L(i,1,:) = LLR;
    end

    % Path Metric
    PM = zeros(l,1);

    % start  at  root
    node = 0; depth = 0;

    % stopping criteria
    done = 0; counter = 1;
    % traverse till all bits are decoded
```

Iteration Starting for Decoding

```
    while (~(done == 1 && depth == -1))
```

```
        % length of current node
        cur_len = 2^(n-depth);

        % position of node in node state vector
        npos = (2^depth-1) + node + 1;

        node_type_ind = cur_len*node;

        if (depth == n && done == 0)
            % if frozen node
            if (info_check_vec(node+1) == 0)

                ucap(:,n+1,node+1) = 0;
                pen_pos = (L(:,n+1,node+1) < 0);
                PM(pen_pos) = PM(pen_pos) + abs(L(pen_pos,n+1,node+1));
                [PM, ind_ord_l] = sort(PM,'ascend');
                ind_ord_mat(:,n+1,node+1) = ind_ord_l;

            elseif ( counter > log2(l) )
                Lin_temp = [L(:,n+1,node+1);L(:,n+1,node+1)];
                PM_temp = [PM; PM];
                codeword_temp = [zeros(l,1); ones(l,1)];
                pen_pos = ( (Lin_temp < 0) ~= codeword_temp );

                PM_temp(pen_pos) = PM_temp(pen_pos) + abs(Lin_temp(pen_pos));

                [PM_temp, ind_ord] = sort(PM_temp,'ascend');

                ucap(:,n+1,node+1) = codeword_temp(ind_ord(1:l));

                PM = PM_temp(1:l);
                ind_ord = ind_ord(1:l);
                ind_ord(ind_ord > l) = ind_ord(ind_ord > l) - l;
                ind_ord_mat(:,n+1,node+1) = ind_ord;
                counter = counter + 1;

            elseif (counter == 1 || counter == 2 || counter == 3 || counter
== 4)

                if (counter == 1)
                    codeword = [zeros(l/2,1); ones(l/2,1)];
                elseif (counter == 2)
                    codeword = [zeros(l/4,1); ones(l/4,1)];
                    codeword = repmat(codeword,2,1);
                elseif (counter == 3)
                    codeword = [zeros(l/8,1); ones(l/8,1)];
                    codeword = repmat(codeword,4,1);
                elseif (counter == 4)
                    codeword = [zeros(l/16,1); ones(l/16,1)];
```

```
                    codeword = repmat(codeword,8,1);
                end

                ucap(:,n+1,node+1) = codeword;
                Lin = squeeze(L(:,n+1,node+1));
                pen_pos = ( (Lin < 0) ~= codeword );
                PM(pen_pos) = PM(pen_pos) + abs(Lin(pen_pos));
                [PM, ind_ord_l] = sort(PM,'ascend');
                ind_ord_mat(:,n+1,node+1) = ind_ord_l;
                ucap(:,n+1,node+1) = codeword(ind_ord_l);
                counter = counter + 1;
            end
            % check for last leaf node
            if node == (N-1)
                done = 1; node = floor(node/2); depth = depth - 1;

                % move back to parent node
            else
                node = floor(node/2); depth = depth - 1;
            end

        % non-leaf node
        else

            % propagate to left child
            if ns(npos) == 0

                % incoming beliefs
                Ln = L(:,depth+1,node_type_ind + 1:node_type_ind + cur_len);

                % next node: left child
                node = 2*node; depth = depth + 1;

                % incoming belief length for left child
                cur_len = floor(cur_len / 2);

                % calculate and store LLRs for left child
                L(:,depth+1,node_type_ind+1:node_type_ind+cur_len) =
f_minsum(Ln(:,1,1:cur_len),Ln(:,1,cur_len+1:end));

                % mark as left child visited
                ns(npos) = 1;
            else
                % propagate to right child
                if ns(npos) == 1

                    % incoming beliefs
                    ind_cur_ord = ind_ord_mat(:,depth+2,node_type_ind + 1);
                    Ln =
L(ind_cur_ord,depth+1,node_type_ind+1:node_type_ind+cur_len);
```

```
                    % left child
                    lnode = 2*node; ldepth = depth + 1;
                    ltemp = cur_len/2;

                    % incoming decisions from left child
                    ucapn = ucap(:,ldepth+1,ltemp*lnode+1:ltemp*(lnode+1));

                    % next node: right child
                    node = node *2 + 1; depth = depth + 1;

                    % incoming belief length for right child
                    cur_len = floor(cur_len / 2);

                    % calculate and store LLRs for right child
                    L(:,depth+1,cur_len*node+1:cur_len*(node+1)) =
g_minsum(Ln(:,1,1:cur_len),Ln(:,1,cur_len+1:end),ucapn);

                    % mark as right child visited
                    ns(npos) = 2;

                % calculate beta propagate to parent node
                else
                    % ordering
                    ind_ord_temp = ind_ord_mat(:, depth+2, node_type_ind +
1);

                    ind_ord_temp2 = ind_ord_mat(:, depth+2, node_type_ind +
1 + floor(cur_len/2));

                    ind_ord_mat(:,depth+1,node_type_ind + 1) =
ind_ord_temp(ind_ord_temp2);

                    % left and right child
                    lnode = 2*node; rnode = 2*node + 1; cdepth = depth + 1;
                    ctemp = cur_len/2;

                    % incoming decisions from left child
                    i_temp_ord = ind_ord_mat(:,cdepth+1,node_type_ind + 1 +
cur_len/2);
                    ucapl = ucap(:,cdepth+1,ctemp*lnode+1:ctemp*(lnode+1));

                    % incoming decisions from right child
                    ucapr = ucap(:,cdepth+1,ctemp*rnode+1:ctemp*(rnode+1));

                    % combine
                    ucap(:,depth+1,node_type_ind+1:node_type_ind+cur_len) =
cat(3, mod(ucapl(i_temp_ord,:,:)+ucapr,2), ucapr);

                    % update to index of parent node
```

```
                    node = floor(node/2); depth = depth - 1;
                end
            end
        end
    end

    if l == 1
        msg_hat = squeeze(ucap(:,n+1,data_pos))';
    else
        msg_hat = squeeze(ucap(:,n+1,data_pos));
    end

end
```

Output of the MATLAB codes :

Polar Encoding and Decoding

Variables :

N : Codeword length
K : Message Bits
Q1: Reliability sequence for N bits
msg : Message bits
input_Seq : input sequence i.e first N-K bits are frozen (0) and next K bits are message bits
c : Codeword sequence after polar encoding
y : Codeword sequence after BPSK mapping
res : decoded sequence

```
Enter the value of N:
8
Enter the value of K:
4
>> Q1

Q1 =

     1     2     3     5     4     6     7     8

>> msg

msg =

     0     1     1     1

>> input_seq

input_seq =

     0     0     0     0     0     1     1     1

>> c

c =

     1     0     0     1     1     0     0     1

>> y

y =

    -1     1     1    -1    -1     1     1    -1

>>


msg =

     0     1     1     1

>> res

res =

     0     0     0     0     0     1     1     1

>> input_seq

input_seq =

     0     0     0     0     0     1     1     1
```