```
% Rel_Seq is the Bit-Channel Reliability Sequence for 1024 bits
Rel_Seq = [1 2 3 5 9 17 33 4 6 65 10 7 18 11 19 129 13 34 66 21 257 35 25 37
8 130 67 513 12 41 69 131 ...
    20 14 49 15 73 258 22 133 36 259 27 514 81 38 26 23 137 261 265 39 515
97 68 42 145 29 70 43 ...
    517 50 75 273 161 521 289 529 193 545 71 45 132 82 51 74 16 321 134 53
24 135 385 77 138 83 57 28 ...
    98 40 260 85 139 146 262 30 44 99 516 89 141 31 147 72 263 266 162 577
46 101 641 52 149 47 76 267 274 518 105 163 ...
    54 194 153 78 165 769 269 275 519 55 84 58 522 113 136 79 290 195 86 277
523 59 169 140 100 87 61 281 90 291 530 525 ...
    197 142 102 148 177 143 531 322 32 201 91 546 293 323 533 264 150 103
106 305 297 164 93 48 268 386 547 325 209 387 151 154 ...
    166 107 56 329 537 578 549 114 155 80 270 109 579 225 167 520 553 196
271 642 524 276 581 292 60 170 561 115 278 157 88 198 ...
    117 171 62 532 526 643 282 279 527 178 294 389 92 585 770 199 173 121
202 337 63 283 144 104 179 295 94 645 203 593 324 393 ...
    298 771 108 181 152 210 285 649 95 205 299 401 609 353 326 534 156 211
306 548 301 110 185 535 538 116 168 226 327 307 773 158 ...
    657 330 111 118 213 172 777 331 227 550 539 388 309 217 417 272 280 159
338 551 673 119 333 580 541 390 174 122 554 200 785 180 ...
    229 339 313 705 391 175 555 582 394 284 123 449 354 562 204 64 341 395
528 583 557 182 296 286 233 125 206 183 644 563 287 586 ...
    300 355 212 402 186 397 345 587 646 594 536 241 207 96 328 565 801 403
357 308 302 418 214 569 833 589 187 647 405 228 897 595 ...
    419 303 650 772 361 540 112 332 215 310 189 450 218 409 610 597 552 651
230 160 421 311 542 774 611 658 334 120 601 340 219 369 ...
    653 231 392 314 451 543 335 234 556 775 176 124 659 613 342 778 221 315
425 396 674 584 356 288 184 235 126 558 661 617 343 317 ...
    242 779 564 346 453 398 404 208 675 559 786 433 358 188 237 665 625 588
781 706 127 243 566 399 347 457 359 406 304 570 245 596 ...
    190 567 677 362 707 590 216 787 648 349 420 407 465 681 802 363 591 410
571 789 598 573 220 312 709 599 602 652 422 793 803 612 ...
    603 411 232 689 654 249 370 191 365 655 660 336 481 316 222 371 614 423
426 452 615 544 236 413 344 373 776 318 223 427 454 238 ...
    560 834 805 713 835 662 809 780 618 605 434 721 817 837 348 898 244 663
455 319 676 619 899 782 377 429 666 737 568 841 626 239 ...
    360 458 400 788 592 679 435 678 350 246 459 667 621 364 128 192 783 408
437 627 572 466 682 247 708 351 600 669 791 461 250 683 ...
    574 412 804 790 710 366 441 629 690 375 424 467 794 251 372 482 575 414
604 367 469 656 901 806 616 685 711 430 795 253 374 606 ...
    849 691 714 633 483 807 428 905 415 224 664 693 836 620 473 456 797 810
715 722 838 717 865 811 607 913 723 697 378 436 818 320 ...
    622 813 485 431 839 668 489 240 379 460 623 628 438 381 819 462 497 670
680 725 842 630 352 468 439 738 252 463 443 442 470 248 ...
    684 843 739 900 671 784 850 821 729 929 792 368 902 631 686 845 634 712
254 692 825 903 687 741 851 376 445 471 484 416 486 906 ...
    796 474 635 745 853 961 866 694 798 907 716 808 475 637 695 255 718 576
914 799 812 380 698 432 608 490 867 724 487 909 719 814 ...
```

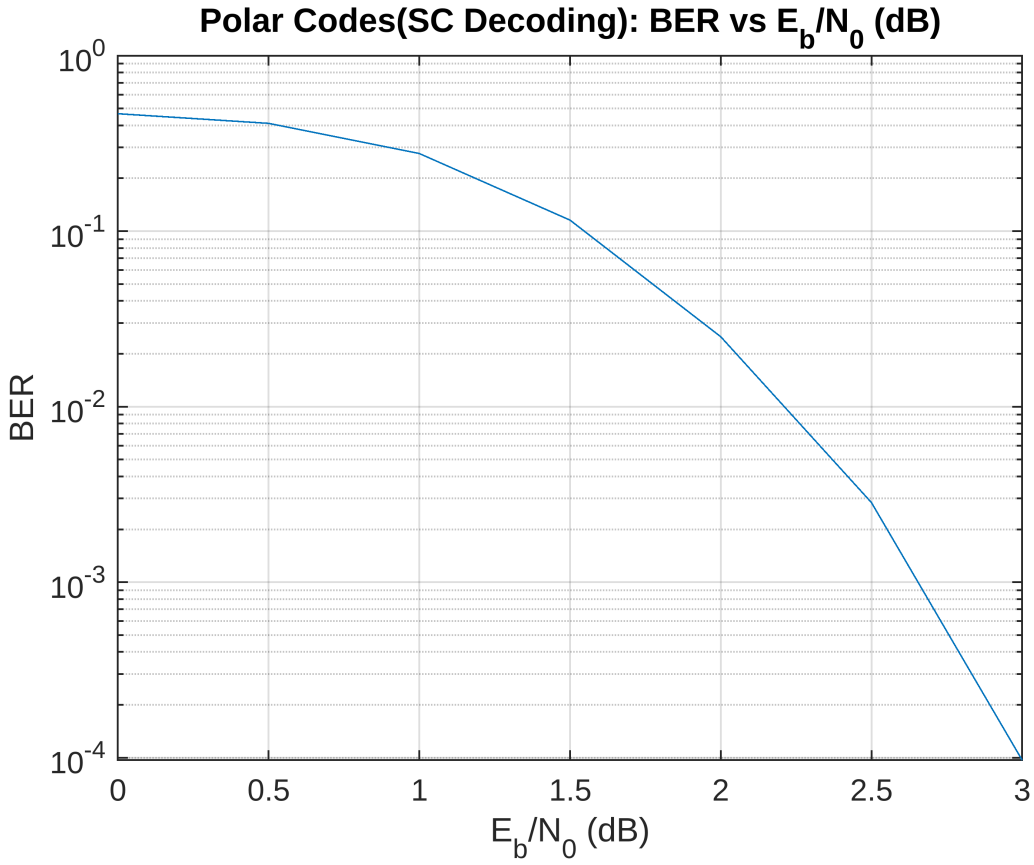
```
    477 857 840 726 699 915 753 869 820 815 440 930 491 624 672 740 917 464
844 382 498 931 822 727 962 873 493 632 730 701 444 742 ...
    846 921 383 823 852 731 499 881 743 446 472 636 933 688 904 826 501 847
746 827 733 447 963 937 476 854 868 638 908 488 696 747 ...
    829 754 855 858 505 800 256 965 910 720 478 916 639 749 945 870 492 700
755 859 479 969 384 911 816 977 871 918 728 494 874 702 ...
    932 757 861 500 732 824 923 875 919 503 934 744 761 882 495 703 922 502
877 848 993 448 734 828 935 883 938 964 748 506 856 925 ...
    735 830 966 939 885 507 750 946 967 756 860 941 831 912 872 640 889 480
947 751 970 509 862 758 971 920 876 863 759 949 978 924 ...
    973 762 878 953 496 704 936 979 884 763 504 926 879 736 994 886 940 995
981 927 765 942 968 887 832 948 508 890 985 752 943 997 ...
    972 891 510 950 974 1001 893 951 864 760 1009 511 980 954 764 975 955
880 982 983 928 996 766 957 888 986 998 987 944 892 999 767 ...
    512 989 1002 952 1003 894 976 895 1010 956 1005 1011 958 984 959 988
1013 1000 1017 768 990 1004 991 1006 960 1012 1014 896 1007 1015 1018 1019
...
    992 1021 1008 1016 1020 1022 1023 1024];

G = [1, 0; 1, 1]; % Polar Transform
N = 1024; % Length of Code-word
K = 512; % Length of Message Bits
Rate = K/N; % Rate
n = log2(N);
EbNodB = 0:0.5:10; % Value of Eb/No in dB
BER = zeros(1, size(EbNodB, 2)); % Storing BER values for plotting against
Eb/No(dB)
Pc = zeros(1, size(EbNodB, 2)); % Storing Probability of Successful Decoding
for plotting against Eb/No(dB)
ind=1;

for i = EbNodB
    EbNo = 10^(i/10); % Value of Eb/No
    sigma = 1/(sqrt(2*Rate*EbNo));

    % To find Polar Transform for N bits
    GN = Nbit_PolarTransform(G, n);

    Rel_SeqN = Rel_Seq(Rel_Seq<=N); % Reliability Sequence for N bits

    Frozen = Rel_SeqN(1:N-K); % Frozen Positions

    % Simulating the Code
    F_ni = 0;
    BitErrors=0;
    Nsim=10000;

    for Block = 1:Nsim
        message = randi([0 1], 1, K); % K-bit message
```

```
        u = zeros(1, N);

        u(Rel_SeqN(N-K+1:end)) = message; % assigning message bits

        code_word = mod((u*GN), 2); % generating code word

        s = BPSK_Modulation(code_word); % BPSK Modulation on the codeword
        r = AWGN_Channel(s, sigma, N); % Passing through AWGN Channel

        % Successive Cancelation Decoder

        L = r; % Taking the r as beliefs values for our decoder
        node = 1:N;
        [u_cap, x_cap] = Polar_Decode(L, Frozen, node); % Decoding the
received signal

        message_cap = u_cap(Rel_SeqN(N-K+1:end)); % Extracting the message
from ucap

        % Finding total Bits with Error
        errors = sum(message ~= message_cap);
        BitErrors = BitErrors + errors;
        if (errors==0)
            F_ni = F_ni + 1;
        end
    end

    BER_simulation = BitErrors/(K*Nsim);
    BER(ind) = BER_simulation; % Storing the values of BER_simulation for a
given Eb/No(dB)
    Pc(ind) = F_ni/Nsim; % Storing the value of Probability of Successful
decoding for a given Eb/No(dB)
    ind = ind+1;
    format short g;
    disp([i BER_simulation BitErrors (K*Nsim)]);
end
```

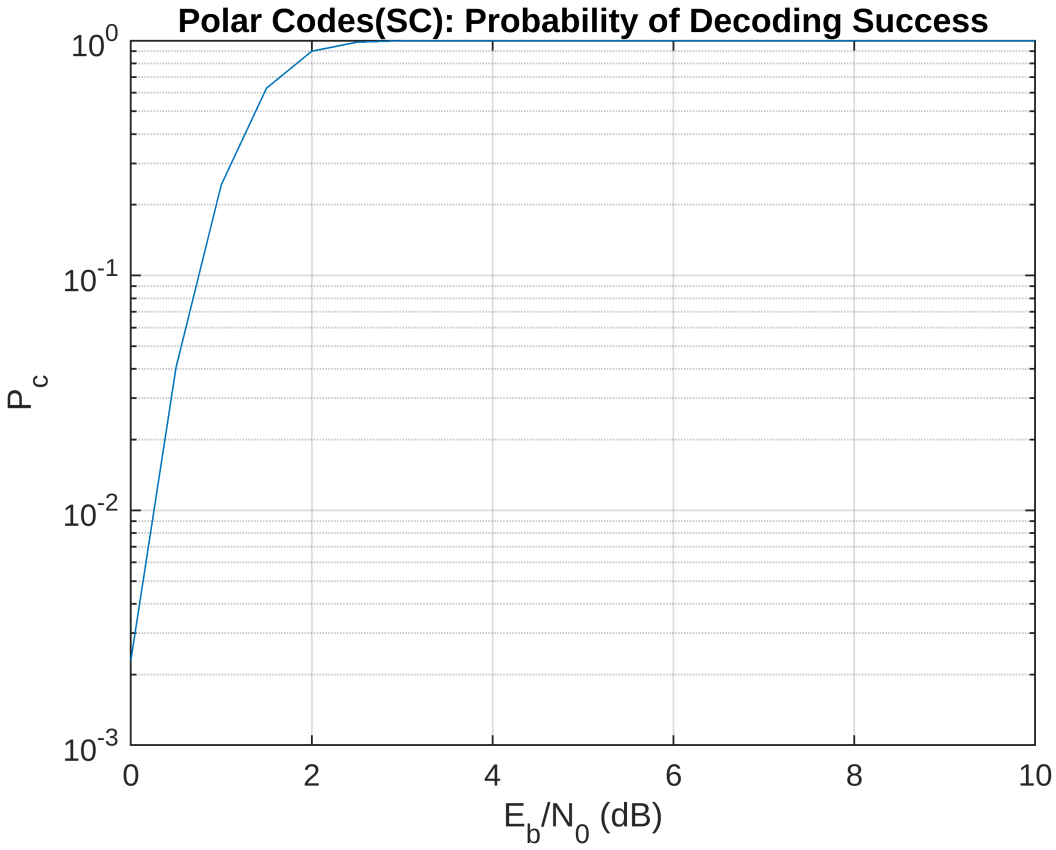
```
          0      0.46663   2.3891e+06     5.12e+06

        0.5      0.41123   2.1055e+06     5.12e+06

          1      0.27678   1.4171e+06     5.12e+06

        1.5       0.1155   5.9136e+05     5.12e+06

          2     0.025015   1.2808e+05     5.12e+06

        2.5    0.0028389        14535     5.12e+06

          3    9.668e-05          495     5.12e+06

        3.5            0            0     5.12e+06

         4           0           0     5120000
```

```
4.5           0           0     5.12e+06
 5            0           0     5120000
5.5           0           0     5.12e+06
 6            0           0     5120000
6.5           0           0     5.12e+06
 7            0           0     5120000
7.5           0           0     5.12e+06
 8            0           0     5120000
8.5           0           0     5.12e+06
 9            0           0     5120000
9.5           0           0     5.12e+06
10            0           0     5120000
```

```
semilogy(EbNodB, BER); % Plotting BER vs Eb/No(dB)
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('BER');
title('Polar Codes(SC Decoding): BER vs E_b/N_0 (dB)');
```




```
semilogy(EbNodB, Pc); % Plotting Probability of Decoding Success
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('P_c');
title('Polar Codes(SC): Probability of Decoding Success');
```




```
function [u_cap, x_cap] = Polar_Decode(L, Frozen, node)
    N = size(L, 2);
    if (N==1)
        if any(Frozen==node) % To check if node is frozen
                x_cap = 0;
                u_cap = 0;
        else
            if L >= 0
                x_cap = 0;
                u_cap = 0;
            else
                x_cap = 1;
                u_cap = 1;
            end
        end
    else
        % Partitioning Beliefs into two parts
        L1 = L(1:N/2);
        L2 = L(N/2+1:N);
        % Finding the Beliefs for Left Part by SPC Decoding
        L_Left = SPC_Decode(L1, L2);

        % Recursively calling function to get u1_cap and x1_cap
        [u1_cap, x1_cap] = Polar_Decode(L_Left, Frozen, node(1:N/2));
```

```
        % Finding the Beleifs for Right Part by Repetition Decoding
        L_Right = Rep_Decode(L1, L2, x1_cap);

        % Recursively calling function to get u2_cap and x2_cap
        [u2cap, x2cap] = Polar_Decode(L_Right, Frozen, node(N/2+1:N));
        x_cap = [mod((x1_cap+x2cap), 2) x2cap];
        u_cap = [u1_cap u2cap];
    end
end

function C = KroneckerProduct(A, B)
    [m, n] = size(A);
    [p, q] = size(B);
    C = zeros(m*p, n*q);
    for i=1:m
        for j=1:n
            % rows from r1 to r2
            r1 = 1 + p*(i-1);
            r2 = p*i;

            % columns from c1 to c2
            c1 = 1 + q*(j-1);
            c2 = q*j;
            C(r1:r2, c1:c2) = A(i, j)*B;
        end
    end
end

function GN = Nbit_PolarTransform(G, n)
    GN = G;
    for i = 1:n-1
        GN = KroneckerProduct(G, GN);
    end
end

function L = SPC_Decode(a, b)
    % Calculating the sign of 'a' and 'b' respectively
    signA = (1-2*(a<0));
    signB = (1-2*(b<0));

    % Calculating the minimum of the absolute values of 'a' and 'b'
    minAbs = min(abs(a), abs(b));

    % Calculating the final result
    L = signA.*signB.*minAbs;
end

function L = Rep_Decode(a, b, c)
    % c=0 --> L = b+a
```

```
    % c=1 --> L = b-a
    L = b + (1-2*c).*a;
end

function s = BPSK_Modulation(code_word)
    % 0 --> 1       1 --> -1
    s = 1 - (2*code_word);
end

function r = AWGN_Channel(s, sigma, N)
    % Additive White Gaussian Distributed Noise added to symbols
    r = s + sigma*randn(1, N);
end
```