```
// find the zeroes of legendre polynomial n.

/*
P-1(x)=0
P0(x)=1
Pk(x)=(2k-1)/k*xPk-1(x)-(k-1)/k*Pk-2(x)

polynomial k has k roots, those roots are interleaved with the roots
of Pk-1.
*/

#include<iostream>
#include<math.h>


// #define K 2 // this is the legendre poly whose roots we will find.
#define ACC .000001 // accuracy of our newton iteration method.

double funcEval(int termK, double x){

        if(termK == 0)
                return 1;
        if(termK == 1)
                return x;

        double cur, k_1, k_2;
        k_1=1;
        k_2=0;
        for(int k = 1; k <= termK; k++){
                cur = (2.*(double)k-1.)/(double)k*x*k_1-((double)k-
1.)/(double)k*k_2;
//              printf("funcEval k cur: %i %f\n",k,cur);
                k_2 = k_1;
                k_1 = cur;
        }
//      printf("returning %f\n",cur);
        return cur;
}

double derivEval(int termK, double x){

        if(termK == 0)
                return 0;
        if(termK == 1)
                return 1;

        double cur, k_1, k_2;
        k_1=1;
        k_2=0;
        for(int k = 2; k <= termK; k++){
                cur = (2.*(double)k-1.)/(double)k*(funcEval(k-
1,x)+x*k_1) - ((double)k-1.)/(double)k*k_2;
//              printf("derivEval k cur: %i %f\n",k,cur);
                k_2 = k_1;
                k_1 = cur;
        }
        return cur;
}

double newton(double guess, int K){
```

```
        // finds the zero of legendre poly of order tempK given by
funcEval, derivEval,
        // closest to guess

        double guessN = guess, guessN_1 = 1000000000.,temp;

        while(fabs(guessN-guessN_1) > ACC){
//              printf("error: %f, newGuess: %f, lastGuess:
%f\n",fabs(guessN-guessN_1),guessN,guessN_1);
                guessN_1 = guessN;
//              printf("K: %i, last guess: %f, func: %f, deriv:
%f\n",K,guessN_1, funcEval(K,guessN_1), derivEval(K,guessN_1));
                guessN = guessN -
funcEval(K,guessN)/derivEval(K,guessN);
//              printf("new Guess: %f\n",guessN);
        }

        //printf("root: %f\n",guessN);
        return guessN;
}

double bisection(double lower, double upper, int K){

        double dynLower = lower;
        double dynUpper = upper;
        double upperFunc, mid;

        if((funcEval(K,lower) > 0 && funcEval(K,upper) > 0) ||
(funcEval(K,lower) < 0 && funcEval(K,upper) < 0))
                printf("Error: upper and lower have same sign\n");

        double guess;
        while(fabs(dynLower - dynUpper) > ACC){
                mid = (dynLower+dynUpper)/2.;
                guess = funcEval(K,mid);
                if(guess == 0.)
                        return guess;
                upperFunc = funcEval(K,dynUpper);
                if((guess > 0 && upperFunc > 0) || (guess < 0 &&
upperFunc < 0))
                        dynUpper = mid;
                else
                        dynLower = mid;
                //printf("dynLower: %f, dynUpper: %f, guess:
%f\n",dynLower, dynUpper, guess);
        }

        //printf("bisection finished\n");
        return mid;
}

int main(int argc,char** argv) {


        int K = 105; // this is the legendre poly whose roots we will
find.

//      funcEval(K,2);
//      printf("%f\n",derivEval(K,-1./3.));
```

```
	// I'll try to see if evenly spacing our initial guesses over the interval [-1,1]
	// is good enough to find all the roots of each poly using newton iterations.

	double guess;
	double roots[K];
	double nextRoots[K];
	roots[0] = 0;

	for(int i = 1; i < K; i++){
		for(int j = 0; j < i+1; j++){
			//guess = -1. + (2./((double)K+1.)+i*(2./((double)K+1.)));

			/*if(j == 0)
				guess = .5 * (-1. + roots[0]);
			else if(j > 0 && j < i)
				guess = .5 * (roots[j-1] + roots[j]);
			else
				guess = .5 * (1. + roots[j-1]);

			// constructs guess as the mean between either the boundaries or the roots on either
			// side of the current root we're looking for.

			//printf("K: %i, guess: %f\n",i+1,guess);

			nextRoots[j] = newton(guess,i+1);*/

			if(j == 0)
				nextRoots[j] = bisection(-1.,roots[0],i+1);
			else if(j > 0 && j < i)
				nextRoots[j] = bisection(roots[j-1],roots[j],i+1);
			else
				nextRoots[j] = bisection(roots[j-1],1.,i+1);


		}
		for(int j = 0; j < K; j++){
			roots[j] = nextRoots[j]; // copy the roots over from the new generated set over the old set.
			//printf("K: %i, roots[%i]: %f\n",i+1,j,roots[j]);
		}
		//printf("iteration %i complete\n",i);
	}

	for(int i = 0; i < K; i++)
		printf("roots[%i]: %f\n",i,roots[i]);


	return 0;
}
```

n	x_j	w_j
2	$\pm\frac{1}{3}\sqrt{3}$	1
3	0	$\frac{8}{9}$
	$\pm\frac{1}{5}\sqrt{15}$	$\frac{5}{9}$
4	$\pm\frac{1}{35}\sqrt{525-70\sqrt{30}}$	$\frac{1}{36}\left(18+\sqrt{30}\right)$
	$\pm\frac{1}{35}\sqrt{525+70\sqrt{30}}$	$\frac{1}{36}\left(18-\sqrt{30}\right)$
5	0	$\frac{128}{255}$
	$\pm\frac{1}{21}\sqrt{245-14\sqrt{70}}$	$\frac{1}{900}\left(322+13\sqrt{70}\right)$
	$\pm\frac{1}{21}\sqrt{245+14\sqrt{70}}$	$\frac{1}{900}\left(322-13\sqrt{70}\right)$

N	x_j	w_j	x_j	w_j
2	$\pm\frac{1}{3}\sqrt{3}$	1	±0.57735 02691	1
3	0	$\frac{8}{9}$	0	0.88888 88888
	$\pm\frac{1}{5}\sqrt{15}$	$\frac{5}{9}$	±0.77495 66692	0.55555 55555
4	$\pm\frac{1}{35}\sqrt{525-70\sqrt{30}}$	$\frac{1}{36}\left(18+\sqrt{30}\right)$	±0.33998 10435	0.65214 51548
	$\pm\frac{1}{35}\sqrt{525+70\sqrt{30}}$	$\frac{1}{36}\left(18-\sqrt{30}\right)$	±0.86113 63115	0.34785 48451
5	0	$\frac{128}{255}$	0	0.56888 88888
	$\pm\frac{1}{21}\sqrt{245-14\sqrt{70}}$	$\frac{1}{900}\left(322+13\sqrt{70}\right)$	±0.53846 93101	0.47862 86704
	$\pm\frac{1}{21}\sqrt{245+14\sqrt{70}}$	$\frac{1}{900}\left(322-13\sqrt{70}\right)$	±0.90617 98459	0.23692 68850