Software test of Newton-Raphson square root.

```
// Code for pulling out the exponent of the float.
int float_exponent(float Input)
{
	union {
		float y;
		long  k;
	} temp;
	temp.y = Input;

	return ((int)((temp.k & 0x7f800000) >> 23) - 127);
}

// Code for "computing" pow( 2.0, exponent )
float Two2exponent(long exponent)
{
	union {
		float y;
		long  k;
	} temp;

	temp.k = (((exponent + 127) & 0xff) << 23);
	return temp.y;
}


// Classic Newton Rasphon approach to computing square root
// with debug prints.
float NR_sqrt_debug(float b)
{
	if (b > 0.0) // Only positive numbers.
	{
		int exponent; // Exponent drawn from float.
		double xi, xi_1; // Iteration variables.
		xi = 0.0;
		// initial guess is pow( 2, exponent/2 )
		exponent = float_exponent(b);
		Serial.print("Exponent = ");
		Serial.println(exponent);
		xi_1 = Two2exponent(exponent / 2);

		// iterate until no change.
		while (xi != xi_1)
		{
			Serial.println(xi_1, 7);
			xi = xi_1; // save last value
			xi_1 = 0.5 * (xi + b / xi); // step via NR.
		}

		return xi;
	}
	return log(-1); // Send NaN for negative numbers.
}
```

```
// Classic Newton Rasphon approach to computing square root.
float NR_sqrt(float b)
{
	if (b > 0.0) // Only positive numbers.
	{
		int exponent; // Exponent drawn from float.
		double xi, xi_1; // Iteration variables.
		xi = 0.0;
		// initial guess is pow( 2, exponent/2 )
		exponent = float_exponent(b);
		xi_1 = Two2exponent(exponent / 2);

		// iterate until no change.
		while (xi != xi_1)
		{
			xi = xi_1; // save last value
			xi_1 = 0.5 * (xi + b / xi); // step via NR.
		}
		return xi;
	} // End of positive test.
	return log(-1); // Send NaN for negative numbers.

} // End of NR_sqrt

unsigned long Timer;
// put your setup code here, to run once:
void setup()
{
	float x, y,        // Working variables.
		err = 0.0; // holds maximum error
	Serial.begin(9600);
	// First we run the deubg version
	// that prints out the iterations.
	y = NR_sqrt_debug(5);
	Serial.print("Relative Error for NR_sqrt is ");
	Serial.println(y*y - 5.0, 7);

	// Timer to time off calculations
	Timer = micros();
	// Loop through values of
	for (x = 1.0; x < 100000; x++)
	{
		y = NR_sqrt(x);
		err = max( abs(y*y - x), err);
	}
	Timer = micros() - Timer;

	Serial.print("Maximum relative error for NR_sqrt =  ");
	Serial.println(err, 7);
	Serial.print("Time Required = ");
	Serial.print(1e-6*Timer);
	Serial.println(" Seconds");

	Timer = micros();
	for (x = 1.0; x < 100000; x++)
	{
		y = sqrt(x);
		err = max( abs(y*y - x), err);
	}
	Timer = micros() - Timer;

	Serial.print("Maximum relative error for sqrt = ");
	Serial.println(err, 7);
	Serial.print("Time Required = ");
	Serial.print(1e-6*Timer);
	Serial.println(" Seconds");
```

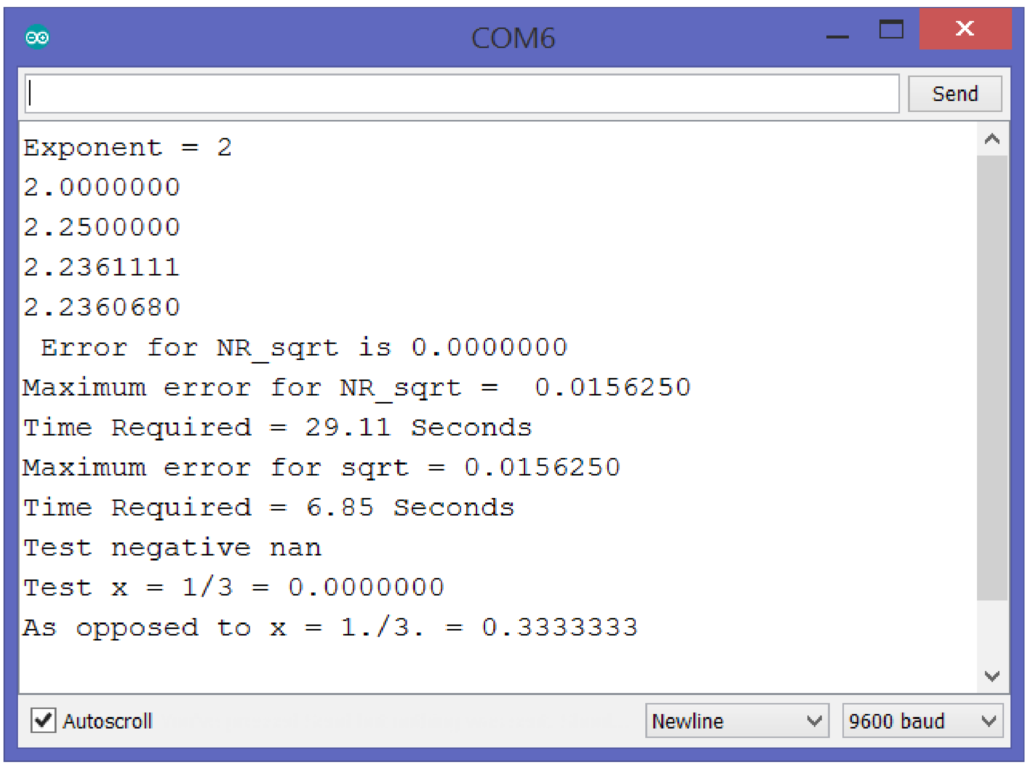
```
        // Test some special cases.
        Serial.print("Test negative ");
        Serial.println(NR_sqrt(-1.0));

        Serial.print("Test x = 1/3 = ");
        x = 1 / 3;
        Serial.println(x, 7);
        Serial.print("As opposed to x = 1./3. = ");
        x = 1. / 3.;
        Serial.println(x, 7);

} // End of setup

// put your main code here, to run repeatedly:
void loop()
{
        // Nothing here.
}
```

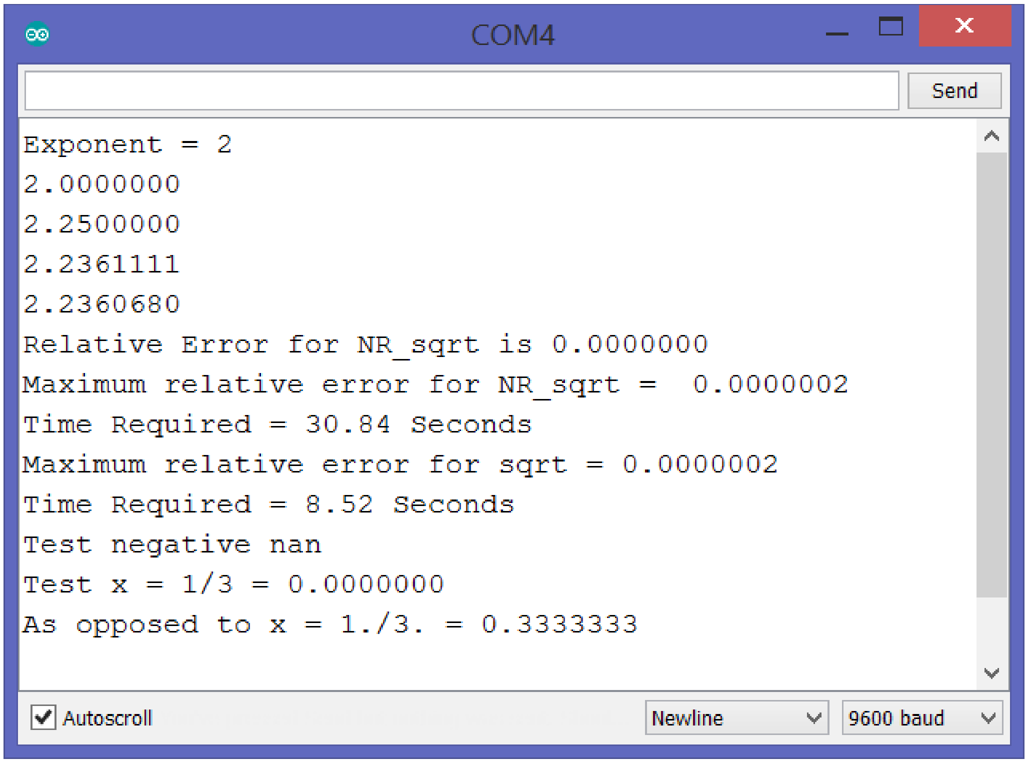
Results of Previous Code.



Simply changing the error calculations to

```
err = max((abs(y*y - x) / x), err);
```

Where it is normalized by the size of x, we have



The error is now shown to be on the order of the precision of floating point numbers.