1. **1a** MATLAB outputs:

   ```matlab
   >> a=(1+3.4e-16)-1.1e-16;a-1
   ans =
       4.4409e-16
   >> a=1+(3.4e-16-1.1e-16);a-1
   ans =
       2.2204e-16
   ``

   And the exact answer for both operations is 2.3e-16.

   The reason why way I is so inaccurate is that the number 1+3.4e-16 is stored on the computer as $(1 + 3.4 \times 10^{-16})(1 + \epsilon_{mach})$ which corresponds to an error of magnitude $10^{16}$ in double precision. Then the floating point operation $(1 + 3.4 \times 10^{-16}) - 1.1 \times 10^{-16}$ is accurate to $((1 + 3.4 \times 10^{-16}) - 1.1 \times 10^{-16})(1 + \epsilon_{mach})$, which also introduces an error of magnitude $10^{16}$. The final step, $a - 1$, doesn’t introduce significant error into this calculation, since the resulting number is on magnitude of $10^{-16}$ so the error is of the magnitude $10^{-32}$. The two operations which introduce error of magnitude $10^{-16}$ result in an inaccurate answer since the real answer is of the same magnitude.

   Way II is more accurate since $(3.4 \times 10^{-16} - 1.1 \times 10^{-16} = 2.3 \times 10^{-16}(1 + \epsilon_{mach})$ only has an error $\approx 10^{-32}$, which is not significant for this calculation. Then $1 \oplus 2.3 \times 10^{-16} = (1 + 2.3 \times 10^{-16})(1 + \epsilon_{mach})$ introduces an error of magnitude $10^{-16}$ which is significant since the final answer is of this magnitude. Once again, the final operation, $a - 1$ does not introduce significant error. While the returned answer is inaccurate because the introduced error was of the same magnitude as the answer, there was only one such step that introduced such a large error, which compared to two such steps in way I, is more accurate.

**1b** MATLAB outputs:

   ```matlab
   >> x = 0.999; a = 0; for j=1:60000, a = a-x^j/j; end; a-log(1-x)
   ans =
       3.0553e-13
   >> x = 0.999; a = 0; for j=60000:-1:1, a = a-x^j/j; end; a-log(1-x)
   ans =
       -8.8818e-16
   ``

   The answer in each case is the error in evaluating the taylor series for $\ln(1-x)$ for $x = 0.999$ which is $\approx -6.9$.

   For way I, the series is evaluated as $a = - \sum_{j=1}^{60000} \frac{0.999^j}{j}$. For way II, $a = - \sum_{j=1}^{60000} \frac{0.999^{60001-j}}{60001 - j}$.

   Looking at way I, the first term evaluated is $-0.999 \approx 1$ and the second term is $\approx -0.5$. Adding these two terms together introduces an error of $-1.5\epsilon_{mach}$ which is of the magnitude $10^{-16}$. Since the partial sum monotonically approaches $-6.9...$, it is always of magnitude 1. This means that adding each term to the series introduces an error of $\approx 10^{-16}$, which compounded over 60000 terms turns into a total error of $\approx 10^{-12}$. The actual observed error
Computing the eigenvalues of a symmetric matrix via the characteristic polynomial:

2. \( a f(x) = 2x \) computed via \( x \oplus x \).

On a machine, this algorithm is \( f(x) = f(l(x) \oplus f(l(x)) \). To prove backwards stability, we want to show that \( \forall x \in X, f(x) = f(\bar{x}) \) for some \( \bar{x} : \|\bar{x} - x\| = O(\epsilon_{mach}) \). To see if this is possible, we set

\[
\bar{x} = (x + (1 + e_1)) \oplus (1 + e_2) = x(1 + e_3) + x(1 + e_3)
\]

where \( |e_{1,2}| \leq \epsilon_{mach} \). It follows that \( e_3 = e_1 + e_2 + e_1 e_2 \). Thus, \( e_3 \leq 2 \epsilon_{mach} + \epsilon^2_{mach} \), and \( \epsilon_{mach} = O(\epsilon_{mach}) \). Thus, the algorithm is backwards stable.

\( b f(x) = 1 + x \) computed via \( 1 \oplus x \).

Set:

\[
\tilde{f}(x) = (1 + x(1 + e_1))(1 + e_2) = 1 + x(1 + e_3) = f(\bar{x})
\]

where \( |e_{1,2}| \leq \epsilon_{mach} \). Ignoring high-order terms (because of their small size), we have

\[ e_3 = e_1 + e_2 + \frac{e_2}{x} \]

Since for choices of \( |x| \ll 1, e_3 \) will blow up, the algorithm is not backwards stable. However to test stability, consider \( \tilde{f}(x) \) and \( f(\bar{x}) \). If we force \( e_3 < \epsilon_{mach} \), then

\[
\frac{||\tilde{f}(x) - f(\bar{x})||}{||f(\bar{x})||} = \frac{e_2 + x(e_1 + e_2 - e_3 - e_1 e_2)}{1 + x(1 + e_3)}
\]

We see that for \( |x| \geq 1 \), the algorithm is backwards stable, so focusing on \( |x| < 1 \), if \( |x| \) is still close to 1, the numerator is \( O(\epsilon_{mach}) \), while the denominator \( \approx 2 \), thus \( \frac{||f(x) - f(\bar{x})||}{||f(\bar{x})||} = O(\epsilon_{mach}) \). For \( |x| \ll 1 \), the numerator is \( \approx \epsilon_2 \) and the denominator is \( \approx 1 \), so \( \frac{||\tilde{f}(x) - f(\bar{x})||}{||f(\bar{x})||} = O(\epsilon_{mach}) \). Thus, while the algorithm is not backwards stable, it is stable.

3. Computing the eigenvalues of a symmetric matrix via the characteristic polynomial:

i Restricting to 2x2 diagonal matrices with diagonal entries \( a \) and \( b \), the characteristic polynomial is \( \lambda^2 - (a + b)\lambda + ab \). If we choose \( A = I \), then we have \( \lambda^2 - 2\lambda + 1 \). Since the coefficients of this polynomial are stored with error \( \epsilon_{mach} \), the polynomial can be written \( (1 + \epsilon_a)\lambda^2 - 2(1 + \epsilon_b)\lambda + (1 + \epsilon_c) \) where \( |\epsilon_{a,b,c}| \leq \epsilon_{mach} \). To find the roots of this polynomial using the quadratic formula, we have

\[
\lambda = \frac{2(1 + \epsilon_b) \pm \sqrt{4(1 + \epsilon_b)^2 - 4(1 + \epsilon_a)(1 + \epsilon_c)}}{2(1 + \epsilon_a)}
\]

By expanding the inside of the square root, we see that the largest order error terms will be \( O(\sqrt{\epsilon_{mach}}) \).
ii While the above was done for $A = I$, it is easy to see that for any quadratic polynomial $ax^2 + bx + c$, perturbation of $O(\epsilon_{mach})$ in the coefficients will lead to a perturbation of $O(\sqrt{\epsilon_{mach}})$ in the roots of a quadratic polynomial by placing the coefficients in the quadratic formula. Thus if $\epsilon_{mach} = 10^{-16}$, then there will be an error of $O(10^{-8})$ in the roots of the polynomial. In general, for a degree $n$ polynomial, we would expect to see $O(n\sqrt{\epsilon_{mach}})$ if a similar method is used. Since $\sqrt{\epsilon}$ goes to 0 slower than $\epsilon$, $\frac{||\bar{f}(x) - f(x)||}{||f(x)||} \geq O(\epsilon_{mach})$ for any $\bar{x}$: $\frac{||\bar{x} - x||}{||x||} = O(\epsilon_{mach})$, and the algorithm is not stable. Since stability is a weaker condition than backwards stable, it is also not backwards stable. Thus, the algorithm is unstable.

4. Code: lagrange.m, lagrange_eq.m.
   The following plot was produced:

![Lagrange basis functions for 12 equally spaced points on [-1,1]](image-url)

Note how for each node, all polynomials except for one go to 0, and the one that does not goes to 1.

5. Code: interp_error.m, hw2fns.m
   The two functions in the following questions look like:
The following two plots used 25 equally spaced points on $[-1, 1]$ to create an interpolated polynomial for each function. The interpolation error $E_n(x) := f(x) - (L_n f)(x)$ is plotted.
Note that the error is at most $10^{-9}$.

Note that the error blows up to 250 at edges.
Sup norm of the interpolation error for $f(x) = \sin(2e^x)$ as a function of $n$ equally spaced points on $[-1, 1]$.

Note how line appears to be roughly linear between $n=1$ and 27. Since this is on a semilog plot, this indicates exponential convergence in this region.

To explain the reverse trend when $n > 27$, we note that $\sup_{x \in [-1, 1], k \in \{0, \ldots, n\}} l_k(x)$ grows exponentially:

Sup norm over all $n$ Lagrange basis functions on $[-1, 1]$ as a function of $n$.

Since the supremum is growing exponentially, the supremum in the error of the sum over the
these functions is going to grow exponentially as $\epsilon_{mach} \times \sup_{x \in [-1,1], k \in \{0,...,n\}} l_k(x)$, following the rules of floating point arithmetic. Note that at $n \approx 27$, the supremum is $\approx 10^6$, so the supremum of the error will be $\approx 10^{-10}$ if $\epsilon_{mach} = 10^{-16}$. This is where the graph of the sup norm of the interpolation error above suddenly reverses course, and begins to be less accurate, as the increase in error due to the sup norm of the basis functions overtakes the decrease in error due to adding more terms to the interpolant. Code: \verb|lbasis_sup.m|

6. Code: \verb|interp_error_cheby.m|, \verb|lagrange_cheby.m|
   After updating the codes to use Chebychev nodes instead of equally spaced nodes, the following plots were produced:

   By inspecting the graph, it is clear that for each $l_k(x)$, $\sup_{x \in [-1,1]} |l_k(x)| \approx 1$. 

   ![Graph of Lagrange basis functions for 12 Chebychev points on \([-1,1]\).](image)

   ![Graph of Lagrange basis functions for 12 Chebychev nodes on \([-1,1]\).](image)
Maximum error is $10^{-12}$, which is better than for equally-spaced points.

The maximum error for this function is now $\approx 9 \times 10^{-3}$, which is a drastic improvement over the scenario with equally-spaced nodes.
Sup norm of the interpolation error for $f(x) = \sin(2e^x)$ as a function of $n$ Chebychev points on $[-1, 1]$.

Note that the interpolation error is converging exponentially to 0. Thus, the Runge phenomenon does not persist.

Comparing the graphs for the sup norm of the interpolation error of $f(x) = (1 + 25x^2)^{-1}$ as a function of $n$ Chebychev points on $[-1, 1]$, it appears that the Chebychev points allow a better achievable interpolation error. For equally spaced points, the best possible error is
about $10^{-10}$ while Chebychev points have a best possible error of $10^{-15}$. We can’t reasonably expect the error to get much better, as this is close to $\epsilon_{mach}$ since the maximum for this function is $\approx 1$. 
