1. An integral of the form $\int_0^1 f(x)dx$ was computed by two students using trapezoidal rule with end point corrections involving only the first derivative. The first student reported the value as 0.8 using a grid spacing of h, while the second student reported the value as 0.75 using a grid spacing of $h/2$. A smart, lazy student from NMSC class enters their discussion and gives a better answer of the integral with a higher order of accuracy by processing the information above. How did he do it? What was his answer? What is the order of accuracy of his answer?

Solution:

We obtain a better answer of the integral by the Romberg Integration. It provides a better approximation of the integral by reducing the true error.

$$I = \int_0^1 f(x)dx$$

Trapezoidal rule with end point corrections using the first derivative:

$$\int_{x_0}^{x_n} f(x)dx = \frac{h}{2}\left(f(x_0) + f(x_n)\right) + h\sum_{k=1}^{n-1} f(x_k) - \frac{h^2}{12}\left(f'(x_n) - f'(x_0)\right) + \mathcal{O}(h^4)$$

with step size $h = \dfrac{(x_n - x_0)}{N}$

If $I_1,\ I_2$ are the values of I with sub-intervals of width $h_1,\ h_2$ and $E_1,\ E_2$ their corresponding errors, respectively, then

$$E_1 = -\frac{(x_n - x_0)h_1^4}{480} f^{iv}(y_i) \text{ and } E_2 = -\frac{(x_n - x_0)h_2^4}{480} f^{iv}(y_i)$$

Assuming that f^{iv} is constant regardless of step size, we have

$$\frac{E_1}{E_2} = \frac{h_1^4}{h_2^4}$$

then we have

$$E_1 = \left(\frac{h_1}{h_2}\right)^4 E_2$$

Since $I = I_1 + E_1 = I_2 + E_2$, then

$$I_1 + \left(\frac{h_1}{h_2}\right)^4 E_2 = I_2 + E_2$$

solving for E_2,

$$E_2 = \frac{I_2 - I_1}{(h_1/h_2)^4 - 1}$$

This estimate can then be substituted into $I = I_2 + E_2$ to yield an improved estimate of the integral,

$$I \approx I_2 + \frac{I_2 - I_1}{(h_1/h_2)^4 - 1}$$

It can be shown that the error of this estimate is $\mathcal{O}(h^6)$. Thus, we have combined two trapezoidal rule with end corrections using the first derivative estimates of $\mathcal{O}(h^4)$ to yield a new estimate of $\mathcal{O}(h^6)$.

For the special case where the $h_1 = h$ and $h_2 = h/2$,

$$I \approx I_2 + \frac{I_2 - I_1}{2^4 - 1} = \frac{16}{15} I_2 - \frac{1}{15} I_1$$

Given, $I_1 = 0.8$ using a grid spacing of h and $I_2 = 0.75$ using a grid spacing of $h/2$,

$$I = \frac{16}{15}(0.75) - \frac{1}{15}(0.8) = 0.7467$$

with the order of accuracy $\mathcal{O}(h^6)$.

2. Gaussian quadrature:

- Find the first three monic polynomials (i.e., till quadratic) on $[0,1]$ orthogonal with respect to the inner product

$$\langle f, g \rangle = \int_0^1 \frac{x}{\sqrt{1-x^2}} f(x) g(x) dx$$

Solution:

Given two functions $f, g \in C[0,1]$, we define an inner product of these two functions by

$$\langle f, g \rangle = \int_0^1 w(x) f(x) g(x) dx, \qquad w(x) > 0$$

Thus the definition of the inner product depends on the integration interval $[0,1]$ and a given weight function $w(x)$.

For the above inner product we also have

$$\langle xf, g\rangle_w = \int_0^1 w(x)\ xf(x)g(x)\ dx = \langle f, xg\rangle_w$$

We create a sequence of polynomials $\phi_k(x)$ of degree k for $k = 0, 1, 2, 3, ...$ such that $\langle \phi_i, \phi_j\rangle_w = 0$ for all $i \neq j$.

We know that the Chebyshev polynomials of the first kind $T_n(x)$ are orthogonal within the interval $x \in [-1, 1]$ with a weight function $w(x) = \dfrac{1}{\sqrt{1-x^2}}$,

$$\int_{-1}^{1} \frac{1}{\sqrt{1-x^2}}\ T_i(x)T_j(x)\ dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases}$$

We also know that the Chebyshev polynomials of the first kind $T_n(y)$ are orthogonal polynomials on $[0, 1]$ with respect to the weight function $w(y) = \dfrac{1}{\sqrt{4y-4y^2}}$,

$$\int_{0}^{1} \frac{1}{\sqrt{4y-4y^2}}\ T_i(y)T_j(y)\ dy = \begin{cases} 0 & i \neq j \\ \pi/4 & i = j \neq 0 \\ \pi/2 & i = j = 0 \end{cases}$$

We can transform any finite domain $a \leq y \leq b$ to the basic domain $-1 \leq x \leq 1$ with the change of variable $y = \frac{1}{2}(b-a)x + \frac{1}{2}(b+a)$.
For the domain $0 \leq y \leq 1$, we can write $y = \frac{1}{2}(x+1)$.

Let us now find the sequence of orthogonal polynomials. This is done by a Gram-Schmidt process.

let, $\phi_0(x) = 1 \implies \phi_0(2y-1) = 1$

then, $\phi_1(x) = x - B_1\phi_0(x)$, where $B_1 = \dfrac{\langle x\phi_0, \phi_0\rangle_w}{\|\phi_0\|_w^2}$

$$\begin{aligned}\phi_1(y) &= (2y-1) - \frac{\langle (2y-1)\phi_0, \phi_0\rangle}{\langle \phi_0, \phi_0\rangle}\ \phi_0(2y-1) \\ &= (2y-1) - \frac{\langle (2y-1), 1\rangle}{\langle 1, 1\rangle} \\ &= (2y-1) - \frac{\displaystyle\int_0^1 \frac{1}{\sqrt{4y-y^2}}(2y-1)dy}{\displaystyle\int_0^1 \frac{1}{\sqrt{4y-y^2}}dy} \\ &= y - \frac{1}{2}\end{aligned}$$

and $\phi_k(x) = (x - B_k)\phi_{k-1}(x) - C_k\phi_{k-2}(x), \qquad k \geq 2$

with, $B_k = \dfrac{\langle x\phi_{k-1}, \phi_{k-1}\rangle_w}{\|\phi_{k-1}\|_w^2}$ and

$C_k = \dfrac{\langle x\phi_{k-1}, \phi_{k-2}\rangle_w}{\|\phi_{k-2}\|_w^2} = \dfrac{\|\phi_{k-1}\|_w^2}{\|\phi_{k-2}\|_w^2}$

$$\begin{aligned}
\phi_2(y) &= ((2y-1) - B_2)\ \phi_1(2y-1) - C_2\ \phi_0(2y-1) \\
&= \left((2y-1) - \frac{\langle (2y-1)\phi_1, \phi_1\rangle}{\langle \phi_1, \phi_1\rangle}\right) \phi_1(2y-1) - \frac{\langle (2y-1)\phi_1, \phi_0\rangle}{\langle \phi_0, \phi_0\rangle}\ \phi_0(2y-1) \\
&= (2y-1)^2 - \frac{\langle (2y-1)(2y-1), (2y-1)\rangle}{\langle 2y-1, 2y-1\rangle}\ (2y-1) - \frac{\langle (2y-1)(2y-1), 1\rangle}{\langle 1, 1\rangle} \\
&= (4y^2 - 4y + 1) - \frac{\langle 4y^2 - 4y + 1, 2y-1\rangle}{\langle 2y-1, 2y-1\rangle}\ (2y-1) - \frac{\langle 4y^2 - 4y + 1, 1\rangle}{\langle 1, 1\rangle} \\
&= (4y^2 - 4y + 1) - \frac{\displaystyle\int_0^1 \frac{(4y^2-4y+1)(2y-1)}{\sqrt{4y-4y^2}}dy}{\displaystyle\int_0^1 \frac{(2y-1)(2y-1)}{\sqrt{4y-4y^2}}dy}\ (2y-1) - \frac{\displaystyle\int_0^1 \frac{(4y^2-4y+1)}{\sqrt{4y-4y^2}}dy}{\displaystyle\int_0^1 \frac{1}{\sqrt{4y-4y^2}}dy} \\
&= (4y^2 - 4y + 1) - \frac{\pi/4}{\pi/2} \\
&= y^2 - y + \frac{1}{8}
\end{aligned}$$

- Use the above to find a quadrature formula of the form

$$\int_0^1 \frac{x}{\sqrt{1-x^2}} f(x)dx = \sum_{i=0}^{n} a_i f(x_i)$$

that is exact for all $f(x)$ of degree 3.

Solution:

Let $\langle f, g\rangle_w = \displaystyle\int_a^b w(x)f(x)g(x)\ dx$ (the weighted inner product). By the Gram-Schmidt process, there is a sequence $\{\phi_j\}$ of orthogonal polynomials where ϕ_j has degree j. ϕ_{j+1} has $j+1$ distinct real zeros $x_0, ..., x_n$ in $[a, b]$.

Let $l_i(x)$ be the i-th Lagrange basis polynomial for these zeros and let

$$a_i = \int_a^b l_i(x)\ dx$$

The claim is that with this set of x_i's and a_i's,

$$\int_a^b w(x)f(x)\ dx \approx \sum_{i=0}^{n} a_i f(x_i)$$

has degree $2n + 1$.

Suppose $f \in \mathbb{P}_{2n+1}$. Since p_{n+1} has degree $n+1$, polynomial division gives

$$f = q(x)p_{n+1}(x) + r(x), \qquad\qquad q, r \in \mathbb{P}_n.$$

Plugging this expression into the integral,

$$\begin{aligned} I &= \int_a^b (q(x)p_{n+1}(x) + r(x))w(x)dx \\ &= \langle q, p_{n+1}\rangle_w + \int_a^b r(x)w(x)\ dx \\ &= \int_a^b r(x)w(x)\ dx \end{aligned}$$

because p_{n+1} is orthogonal to all polynomials of degree $\leq n$, which includes q. Now plug the expression into the formula:

$$\begin{aligned} \text{formula} &= \sum_{i=0}^{n} a_i f(x_i) \\ &= \sum_{i=0}^{n} a_i q(x_i)p_{n+1}(x_i) + \sum_{i=0}^{n} a_i r(x_i) \\ &= \sum_{i=0}^{n} a_i r(x_i) \end{aligned}$$

Last, we need to establish that I and the formula are equal. Because $r(x)$ has degree $\leq n$, it is equal to its Lagrange interpolant through the nodes $x_0, ..., x_n$, so

$$r(x) = \sum_{i=0}^{n} r(x_i) l_i(x)$$

Thus, working from the formula for I,

$$I = \int_a^b r(x)w(x)\ dx = \sum_{i=0}^{n} \int_a^b l_i(x)w(x)\ dx = \sum_{i=0}^{n} a_i r(x_i)$$

which establishes equality. To see that the degree of accuracy is exactly $2n+1$, consider

$$f(x) = \prod_{j=0}^{n} (x - x_j)^2$$

Note that the nodes x_i depend on the degree, so really they should be written $x_{n,i}$ (for $i = 0, ..., n$) for ϕ_{n+1}. One can show that, unlike with equally spaced interpolation,

$$\lim_{x\to\infty} |I - \sum_{i=0}^{n} a_i f(x_{n,i})| = 0$$

under reasonable assumptions on f. Thus, Gaussian quadrature does well when adding more points to reduce error when function values of f at any point are available.

In summary, let $\{\phi_j\}$ be an orthogonal basis of polynomials in the inner product $\langle f, g\rangle_w = \int_a^b w(x)f(x)g(x)\ dx$ and let $x_0, ..., x_n$ be the zeros of the polynomial ϕ_{n+1} with Lagrange basis $\{l_k(x)\}$. Then

$$I = \int_a^b w(x)f(x)\ dx \approx \sum_{i=0}^{n} a_i f(x_i), \qquad a_i = \int_a^b l_i(x)w(x)\ dx,$$

called the Gaussian quadrature formula for $w(x)$, has degree of accuracy $2n+1$.

Thus,

$$I = \int_0^1 \frac{x}{\sqrt{1-x^2}} f(x)\ dx = \sum_{i=0}^{1} a_i f(x_i), \qquad a_i = \int_0^1 l_i(x)w(x)\ dx,$$

is exact for all $f(x)$ of degree 3.

- Use the above to evaluate $\int_0^1 \frac{x\sin(x)}{\sqrt{1-x^2}}dx$

Solution:

$$I = \int_0^1 \frac{x\ sin(x)}{\sqrt{1-x^2}}dx \approx \sum_{i=0}^{n} a_i f(x_i) \qquad a_i = \int_0^1 l_i(x)w(x)\ dx$$

where $l_i(x) = \prod_{i\neq j}^{n} \frac{x - x_j}{x_i - x_j}$

The Chebyshev polynomials of the first kind $T_n(y)$ are orthogonal polynomials on $[0, 1]$ with respect to the weight function $w(y) = \frac{1}{\sqrt{4y-4y^2}}$

$$T_2(2y-1) = y^2 - y + \frac{1}{8}$$

with nodes $y_0 = 0.85355339$, $y_1 = 0.14644661$ and the weights $w_0 = w_1 = 0.78539816$

$$I = \int_0^1 \frac{(2y-1)\ sin(2y-1)}{\sqrt{4y-4y^2}}dy \approx \sum_{i=0}^{n} a_i f(2y_i - 1) \qquad a_i = \int_0^1 l_i(2y-1)w(2y-1)\ dy$$

called the Gaussian quadrature formula for $w(y)$ and has degree of accuracy $2n+1$.

For the quadrature formula to be exact for all $f(y)$ of degree $2n+1 = 3 \implies n = 1$,

$$I = \int_0^1 \frac{(2y-1)\ sin(2y-1)}{\sqrt{4y-4y^2}}dy = \sum_{i=0}^{1} a_i f(2y_i - 1)$$

where $a_i = \int_0^1 l_i(2y-1)\ w(2y-1)\ dy$

$$\begin{aligned}
I &= a_0 f(2y_0-1) + a_1 f(2y_1-1) \\
&= \int_0^1 l_0(2y-1)\ w(2y-1)\ dy\ f(2y_0-1) + \int_0^1 l_1(2y-1)\ w(2y-1)\ dy\ f(2y_1-1) \\
&= \int_0^1 \frac{(2y-1)-(2y_1-1)}{(2y_0-1)-(2y_1-1)} \frac{1}{\sqrt{4y-4y^2}}\ dy\ f(2y_0-1) \\
&\quad + \int_0^1 \frac{(2y-1)-(2y_0-1)}{(2y_1-1)-(2y_0-1)} \frac{1}{\sqrt{4y-4y^2}}\ dy\ f(2y_1-1) \\
&= \int_0^1 \frac{(2y-1)-(2(0.14644661)-1)}{2(0.85355339)-1)-(2(0.14644661)-1)} \frac{1}{\sqrt{4y-4y^2}}\ dy\ f(2(0.85355339)-1) \\
&\quad + \int_0^1 \frac{(2y-1)-(2(0.85355339)-1)}{(2(0.14644661)-1)-(2(0.85355339)-1)} \frac{1}{\sqrt{4y-4y^2}}\ dy\ f(2(0.14644661)-1) \\
&= (0.78539816)\ (2(0.85355339)-1)\ sin(2(0.85355339)-1) \\
&\quad + (0.78539816)\ (2(0.14644661)-1)\ sin(2(0.14644661)-1) \\
&= 0.72156522
\end{aligned}$$

3. Comment on using the Newton method to compute the root of the function $f(x) = x^{1/3}$, i.e., if your initial guess is $x_0 = 1$, what would be the value of x_n? Does the method converge to the root we want?

Solution:

Given, $f(x) = x^{1/3}$, then $f'(x) = (1/3)x^{-2/3}$, and the Newton method iteration becomes

$$\begin{aligned}
x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \\
&= x_n - \frac{x_n^{1/3}}{(1/3)x_n^{-2/3}} \\
&= x_n - 3x_n \\
&= -2x_n
\end{aligned}$$

The next 10 estimates are $-2.0,\ 4.0,\ -8.0,\ 16.0,\ -32.0,\ 64.0,\ -128.0,\ 256.9,\ -512.0,\ 1024.0$. It is obvious that things are going bad. In fact, if we start with any non-zero estimate, the estimates oscillate more and more wildly.

If the initial guess is $x_0 = 1$, then the value of $x_n = -2x_{n-1}$. The Newton's method diverges in this case. The tangent line at the root is vertical as in $f(x) = x^{1/3}$.

4. It is given that a sequence of Newton iterates converge to a root r of the function $f(x)$. Further, it is given that the root r is a root of multiplicity 2, i.e., $f(x) = (x-r)^2 g(x)$, where $g(r) \neq 0$. It is also given that the function f, its derivatives till the second order are continuous in the neighbourhood of the root r. If e_n is the error of the n^{th} iterate, i.e., $e_n = x_n - r$, then obtain

$$\lim_{n\to\infty} \frac{e_{n+1}}{e_n}$$

Solution:

Given, $f(x) = (x-r)^2 g(x)$, where $g(r) \neq 0$ then $f'(x) = 2(x-r)g(x) + (x-r)^2 g'(x)$, the Newton's method generates the sequence

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Using the time-honored method of adding and subtracting, we can write this as

$$x_{n+1} - r = x_n - r - \frac{f(x_n)}{f'(x_n)}$$

If we let $e_{n+1} = x_{n+1} - r$ and $e_n = x_n - r$, then we can rewrite the above as

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_n)}$$

$$\begin{aligned}
e_{n+1} &= e_n - \frac{(x_n - r)^2 g(x_n)}{2(x_n - r)g(x_n) + (x_n - r)^2 g'(x_n)} \\
&= e_n - \frac{(x_n - r)g(x_n)}{2g(x_n) + (x_n - r)g'(x_n)} \\
&= e_n - \frac{e_n\ g(x_n)}{2g(x_n) + e_n\ g'(x_n)} \\
&= \frac{2\ e_n\ g(x_n) + e_n^2\ g'(x_n) - e_n\ g(x_n)}{2\ g(x_n) + e_n\ g'(x_n)} \\
&= \frac{(2-1)e_n\ g(x_n) + e_n^2\ g'(x_n)}{2\ g(x_n) + e_n\ g'(x_n)}
\end{aligned}$$

For x_n close to r the term $g'(x_n)$ becomes very small relative to $g(x_n)$, and the Newton iteration reduces to

$$e_{n+1} = \frac{(2-1)e_n\ g(x_n)}{2\ g(x_n)}$$

then Newton's method is locally convergent to r, and the error e_n at step n satisfies

$$\lim_{n\to\infty} \frac{e_{n+1}}{e_n} = \frac{1}{2}$$

5. **Bonus question:** What happens to the above if the root r has a multiplicity m?

Solution:

Let assume that the $(m+1)$ times continuously differentiable function f has a multiplicity m root at r.

Let $f(x) = (x-r)^m g(x)$ where $g(r) \neq 0$ then $f'(x) = m(x-r)^{m-1}g(x) + (x-r)^m g'(x)$, the Newton's method generates the sequence

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Using the time-honored method of adding and subtracting, we can write this as

$$x_{n+1} - r = x_n - r - \frac{f(x_n)}{f'(x_n)}$$

If we let $e_{n+1} = x_{n+1} - r$ and $e_n = x_n - r$, then we can rewrite the above as

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_n)}$$

$$\begin{aligned}
e_{n+1} &= e_n - \frac{(x_n - r)^m g(x_n)}{m(x_n - r)^{m-1}g(x_n) + (x_n - r)^m g'(x_n)} \\
&= e_n - \frac{(x_n - r)^m g(x_n)}{m(x_n - r)^m (x_n - r)^{-1} g(x_n) + (x_n - r)^m g'(x_n)} \\
&= e_n - \frac{(x_n - r) g(x_n)}{m\ g(x_n) + (x_n - r)\ g'(x_n)} \\
&= e_n - \frac{e_n\ g(x_n)}{m\ g(x_n) + e_n\ g'(x_n)} \\
&= \frac{m\ e_n\ g(x_n) + e_n^2\ g'(x_n) - e_n\ g(x_n)}{m\ g(x_n) + e_n\ g'(x_n)} \\
&= \frac{(m-1)e_n\ g(x_n) + e_n^2\ g'(x_n)}{m\ g(x_n) + e_n\ g'(x_n)}
\end{aligned}$$

For x_n close to r the term $g'(x_n)$ becomes very small relative to $g(x_n)$, and the Newton iteration reduces to

$$e_{n+1} = \frac{(m-1)e_n\ g(x_n)}{m\ g(x_n)}$$

then Newton's method is locally convergent to r, and the error e_n at step n satisfies

$$\lim_{n\to\infty} \frac{e_{n+1}}{e_n} = \frac{(m-1)}{m}$$

6. Compute $\int_{-1}^{1} e^{-x^2} dx$ using the

   (a) Trapezoidal rule

   (b) Trapezoidal rule with end corrections using the first derivative

   (c) Trapezoidal rule with end corrections using the first derivative and third derivatives

   (d) Gauss-Legendre quadrature

   - Perform this by subdividing $[-1, 1]$ into $N \in \{2, 5, 10, 20, 50, 100\}$ panels.
   - Plot the decay of the absolute error using the above methods.
   - You may obtain the exact value of the integral up to 20 digits using wolfram alpha.
   - Make sure the figure has a legend and the axes are clearly marked.
   - Ensure that the font size for title, axes, legend are readable.
   - Submit the plots obtained, entire code and the write-up.

Solution:

Given,

$$\int_{-1}^{1} e^{-x^2} dx = \sqrt{\pi}\ \text{erf}(1) \approx 1.49365$$

Trapezoidal rule:

$$\int_{x_0}^{x_n} f(x)dx \approx \frac{h}{2}\Big(f(x_0) + f(x_n)\Big) + h\sum_{k=1}^{n-1} f(x_k)$$

where $h = \dfrac{(x_n - x_0)}{N}$, the grid spacing

Trapezoidal rule with end corrections using the first derivative:

$$\int_{x_0}^{x_n} f(x)dx \approx \frac{h}{2}\Big(f(x_0) + f(x_n)\Big) + h\sum_{k=1}^{n-1} f(x_k) - \frac{h^2}{12}\Big(f'(x_n) - f'(x_0)\Big)$$

Trapezoidal rule with end corrections using the first derivative and third derivatives:

$$\int_{x_0}^{x_n} f(x)dx \approx \frac{h}{2}\Big(f(x_0) + f(x_n)\Big) + h\sum_{k=1}^{n-1} f(x_k) - \frac{h^2}{12}\Big(f'(x_n) - f'(x_0)\Big) + \frac{h^4}{720}\Big(f'''(x_n) - f'''(x_0)\Big)$$

Gauss-Legendre quadrature:

$$\int_{-1}^{1} f(x)dx \approx \sum_{k=1}^{n} w_k\ f(x_k)$$

where n is the number of points, w_k are quadrature weights and x_k are the roots of the n^{th} Legendre polynomial.

Program:

```
#!/usr/bin/env python
# File: program6.py
# Name: D.Saravanan
# Date: 02/12/2021

""" Script to implement quadrature methods and see how the error behaves """

import matplotlib.pyplot as plt
from matplotlib.ticker import ScalarFormatter
import numpy as np

plt.style.use("classic")
plt.rcParams["text.usetex"] = True
plt.rcParams["pgf.texsystem"] = "pdflatex"
plt.rcParams.update(
    {
        "font.family": "serif",
        "font.size": 10,
        "axes.labelsize": 12,
        "axes.titlesize": 12,
        "figure.titlesize": 12,
    }
)

# end points of interval
a, b = -1, 1

# function to be integrated
def f(x):
    return np.exp(-x**2)


# first derivative of the function
def df(x):
    return -2*x*np.exp(-x**2)


# third derivative of the function
def ddf(x):
    return 4*x*(3 - 2*x**2)*np.exp(-x**2)
```

```python


# exact value of the integral
exact = 1.49364826562485405080

# number of grid points
N = [2, 5, 10, 20, 50, 100]
n = np.array(N)

# number of different set of grids
Ngrids = len(N)

h = np.zeros(Ngrids)      # different grid spacings

trap = np.zeros(Ngrids)  # trapezoidal rule
tend = np.zeros(Ngrids)  # trapezoidal rule using first derivative
tent = np.zeros(Ngrids)  # trapezoidal rule using first and third derivative
gleg = np.zeros(Ngrids)  # gauss-legendre quadrature rule

for k, N in enumerate(N):
    h[k] = (b - a) / (N - 1)          # grid spacing
    x = np.linspace(a, b, N)          # grid points

    # nodes and weights calculation of gauss-legendre
    xnode, wnode = np.polynomial.legendre.leggauss(N)

    trap[k] = h[k] * (np.sum(f(x)) - (f(a) + f(b)) / 2)
    tend[k] = trap[k] - (h[k]**2 / 12) * (df(b) - df(a))
    tent[k] = tend[k] + (h[k]**4 / 720) * (ddf(b) - ddf(a))
    gleg[k] = np.inner(wnode, f(xnode))


# error calculations
trap_err = abs(np.double(trap - exact))
tend_err = abs(np.double(tend - exact))
tent_err = abs(np.double(tent - exact))
gleg_err = abs(np.double(gleg - exact))

formatter = ScalarFormatter()
formatter.set_scientific(False)

```

```
fig, ax = plt.subplots()
ax.plot(n, trap_err, "b.--", label=r"$trapezoidal\ rule$")
ax.plot(n, tend_err, "r.--", label=r"$trapezoidal\ rule\ 1st\ derivative$")
ax.plot(n, tent_err, "m.--", label=r"$trapezoidal\ 1st\ \&\ 3rd\ derivative$")
ax.plot(n, gleg_err, "g.--", label=r"$gauss-legendre\ quadrature\ rule$")
ax.set(xlabel=r"$number\ of\ grid\ points$", ylabel=r"$error\ in\ quadrature$")
ax.set_xscale("log", base=2); ax.set_yscale("log", base=10)
ax.xaxis.set_major_formatter(formatter)
ax.set_title(r"$Quadrature\ convergence$")
ax.grid(True); ax.legend(loc="lower left")
plt.savefig("program6.pgf")
```

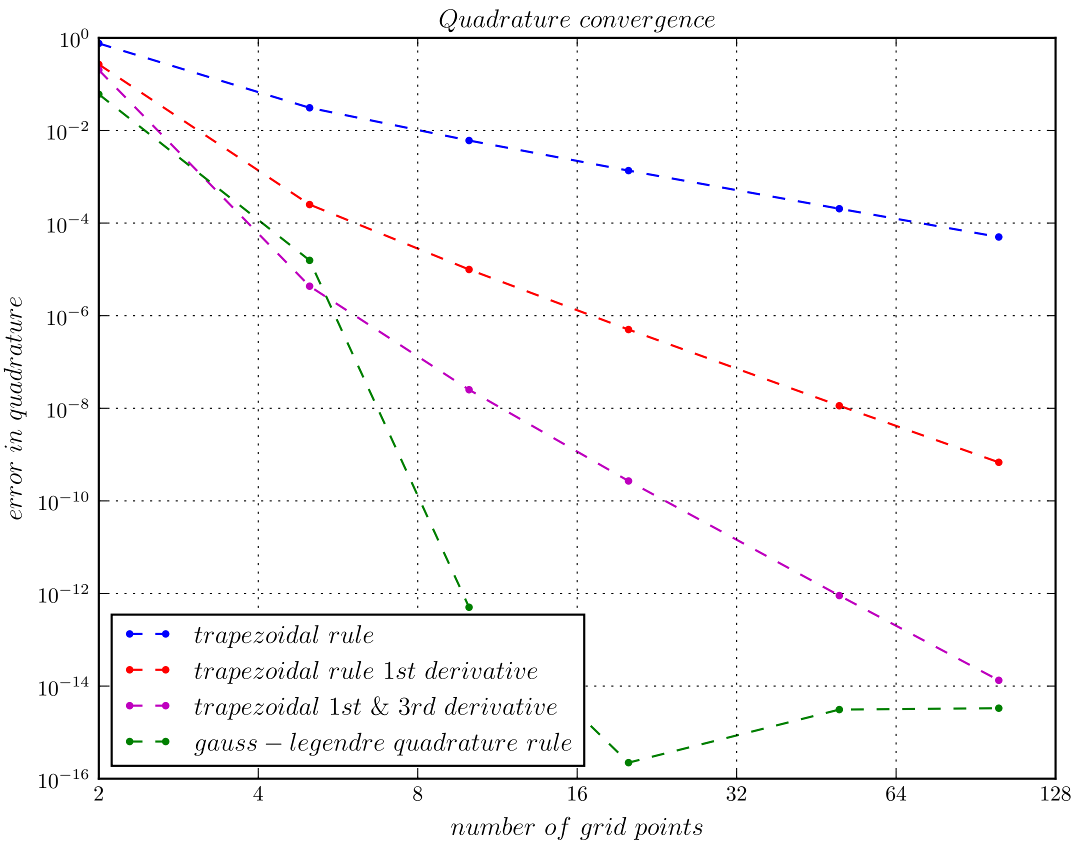


Figure 1: Plot of decay of the absolute error of $\int_{-1}^{1} e^{-x^2} dx$ with $N \in \{2, 5, 10, 20, 50, 100\}$

```
Error terms:
  N         Trap. rule         Trap. 1st       Trap. 1st & 3rd      Gauss-Leg.
  2      7.57889383e-01    2.67383462e-01    2.01982672e-01    6.05856445e-02
  5      3.09077620e-02    2.51141870e-04    4.32996404e-06    1.56550778e-05
 10      6.06557161e-03    9.94294427e-06    2.51687526e-08    5.03375119e-13
 20      1.35924374e-03    5.01574756e-07    2.69072320e-10    2.22044605e-16
 50      2.04303689e-04    1.13439458e-08    9.01945185e-13    3.10862447e-15
100      5.00471987e-05    6.80822287e-10    1.33226763e-14    3.33066907e-15
```

We can see that the Gauss-Legendre quadrature does better than the Trapezoidal rule with end corrections using the first and third derivatives which in turn does better than the Trapezoidal rule with end corrections using the first derivative which in turn does better than the Trapezoidal rule without end corrections.

7. Evaluate $I = \int_0^1 \frac{e^{-x}}{\sqrt{x}}\, dx$ by subdividing the domain into $N \in \{5, 10, 20, 50, 100, 200, 500, 1000\}$ panels.

   (a) Using a rectangular rule

   (b) Make a change of variables $x = t^2$ and use rectangular rule on new variable.

   - Plot the decay of the absolute error using the above two methods.
   - You may obtain the exact value of the integral up to 20 digits using wolfram alpha.
   - Compare the two methods above in terms of accuracy and cost.
   - Explain the difference in solution, if any.
   - Make sure the figure has a legend and the axes are clearly marked.
   - Ensure that the font size for title, axes, legend are readable.
   - Submit the plots obtained, entire code and the write-up.

Solution:

Given,

$$I = \int_0^1 \frac{e^{-x}}{\sqrt{x}}\, dx = \sqrt{\pi}\ \text{erf}(1) \approx 1.493648$$

make a change of variables $x = t^2$, then

$$\frac{dx}{dt} = 2t$$
$$dx = 2t\ dt$$

hence,

$$I = \int_0^1 \frac{e^{-t^2}}{t} 2t\ dt = 2\int_0^1 e^{-t^2} dt = \sqrt{\pi}\ \text{erf}(1) \approx 1.49365$$

Rectangular rule:

$$\int_{x_0}^{x_n} f(x)\ dx \approx h \sum_{k=1}^{n} f\left(\frac{x_{k-1} + x_k}{2}\right)$$

where $h = \dfrac{(x_n - x_0)}{N}$, the grid spacing

Program:

```
#!/usr/bin/env python
# File: program7.py
# Name: D.Saravanan
# Date: 02/12/2021

""" Script to evaluate an integral using rectangular rule """

import matplotlib.pyplot as plt
from matplotlib.ticker import ScalarFormatter
import numpy as np

plt.style.use("classic")
plt.rcParams["text.usetex"] = True
plt.rcParams["pgf.texsystem"] = "pdflatex"
plt.rcParams.update(
    {
        "font.family": "serif",
        "font.size": 10,
        "axes.labelsize": 12,
        "axes.titlesize": 12,
        "figure.titlesize": 12,
    }
)

# end points of interval
a, b = 0, 1

# function to be integrated
def f(x):
    return np.exp(-x)/np.sqrt(x)


# function with chnage of variable
def g(t):
    return 2 * np.exp(-t**2)


```

```
# exact value of the integral
exact = 1.49364826562485405080

# number of grid points
N = [5, 10, 20, 50, 100, 200, 500, 1000]
n = np.array(N)

# number of different set of grids
Ngrids = len(N)

h = np.zeros(Ngrids)         # different grid spacings

rect = np.zeros(Ngrids)      # rectangular rule
recv = np.zeros(Ngrids)      # rectangular rule with change of variables

for k, N in enumerate(N):
    h[k] = (b - a) / (N - 1)         # grid spacing
    x = np.linspace(a, b, N)         # grid points
    y = 0.5*(x[0:N-1] + x[1:N])

    rect[k] = h[k]*np.sum(f(y))
    recv[k] = h[k]*np.sum(g(y))


# error calculations
rect_err = abs(np.double(rect - exact))
recv_err = abs(np.double(recv - exact))

formatter = ScalarFormatter()
formatter.set_scientific(False)

fig, ax = plt.subplots()
ax.plot(n, rect_err, "m.--", label=r"$rectangular\ rule$")
ax.plot(n, recv_err, "b.--", label=r"$rectangular\ rule\ (x = t^2)$")
ax.set(xlabel=r"$number\ of\ grid\ points$", ylabel=r"$error\ in\ quadrature$")
ax.set_xscale("log", base=2); ax.set_yscale("log", base=10)
ax.xaxis.set_major_formatter(formatter)
ax.set_title(r"$Quadrature\ convergence$")
ax.grid(True); ax.legend(loc="lower left")
plt.savefig("program7.pgf")
```

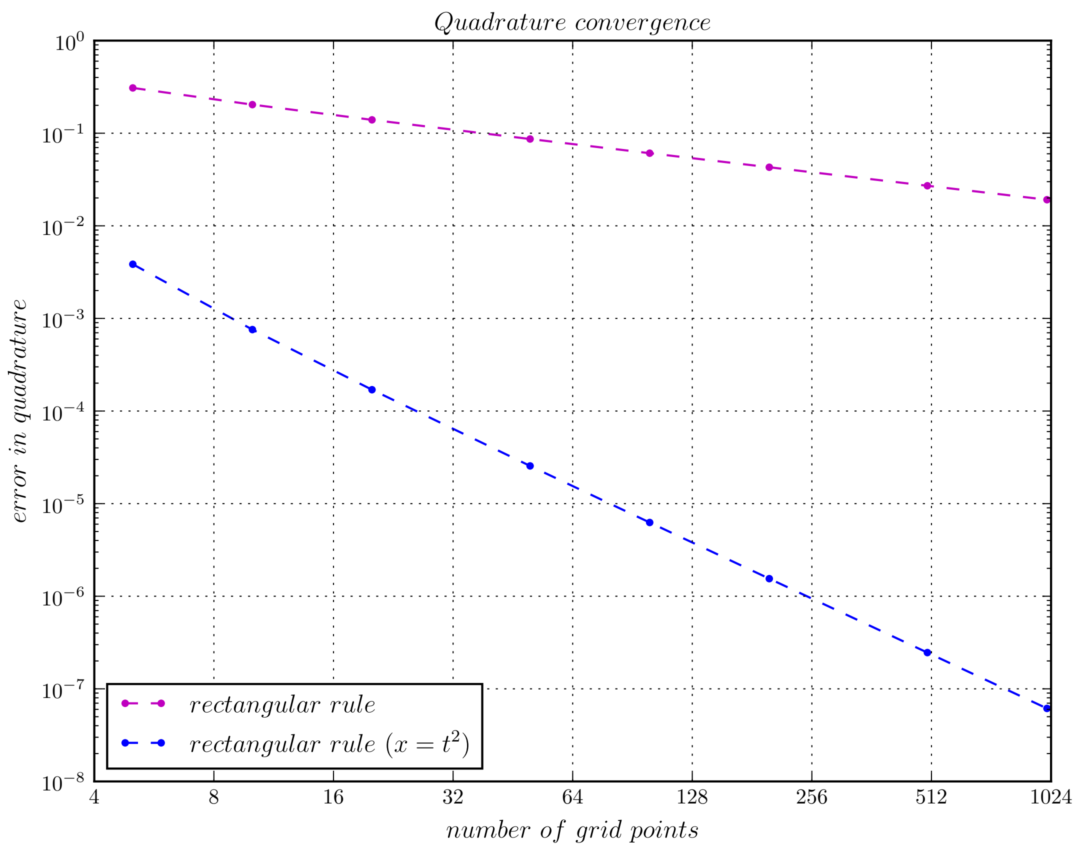
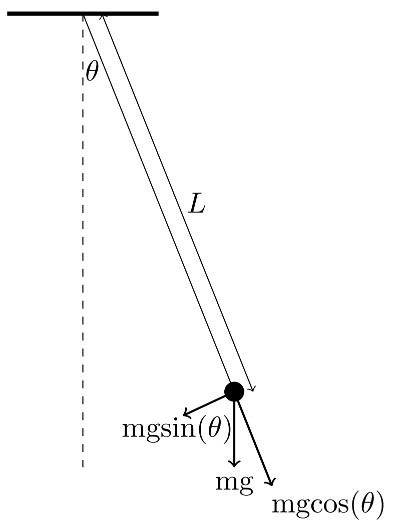


Figure 2: Plot of decay of the absolute error of $\int_0^1 \frac{e^{-x}}{\sqrt{x}}\,dx$ with $N \in \{5, 10, 20, 50, 100, 200, 500, 1000\}$

```
Error terms:
    N        Rect. rule       Rect. (new variable)
    5        0.30836846          3.84599816e-03
   10        0.20357018          7.57498352e-04
   20        0.13943952          1.69870211e-04
   50        0.08658154          2.55371636e-05
  100        0.06085398          6.25585197e-06
  200        0.04290122          1.54827733e-06
  500        0.02708435          2.46236981e-07
 1000        0.01914005          6.14360545e-08
```

We can see that the rectangular rule with change of variables $x = t^2$ does better than the rectangular rule without change of variables. The rectangular rule with change of variables $x = t^2$ generate an integration method with the order of accuracy $\mathcal{O}(h^2)$.

8. Consider the motion of a simple pendulum.



The restoring force is $mg\sin(\theta)$ and hence the governing equation is

$$mL\frac{d^2\theta}{dt^2} + mg\sin(\theta) = 0$$

Let the length of the string be g. Hence, the governing equation simplifies to

$$\frac{d^2\theta}{dt^2} + \sin(\theta) = 0$$

At the initial time, the pendulum is pulled to an angle of $\theta = 30^\circ = \dfrac{\pi}{6}$ before being let loose without any velocity imparted. Write a code to solve for the motion of the pendulum till $t = 100$ seconds using

(a) Forward Euler

(b) Backward Euler

(c) Trapezoidal Rule

- Recall that you need to reformulate the second order differential equation as a system of first order differential equation.
- Vary your time step Δt in $\{0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20\}$.
- For each Δt plot the solution obtained by the three methods on a separate figure till the final time of 100.
- Discuss the stability of the schemes. From your plots, at what Δt do these schemes become unstable (if at all they become unstable)?

- Analyse the stability of the three numerical methods to solve the differential equation by approximating $\sin(\theta)$ to be θ.
- Make sure each figure has a legend and the axes are clearly marked.
- Ensure that the font size for title, axes, legend are readable.
- Submit the plots obtained, entire code and the write-up.

Solution:

We first reduce the second order differential equation

$$\frac{d^2\theta}{dt^2} = -\sin(\theta)$$

to a system of first order equations,

$$\frac{d\theta}{dt} = \omega$$

$$\frac{d\omega}{dt} = -\sin(\theta)$$

Notice that (θ', ω') is given as a function of (θ, ω). The entire motion of the pendulum is determined if we know (θ, ω) at some instant. So we call (θ, ω) the phase of the system. We are given the initial phase of the system, *i.e.,* we know from which initial angle we have released the pendulum, and with what angular velocity. Our aim is to know the phase at all time points during the swing.

Thus, at $t = t_0$, we know

$$\theta = \theta_0 = \frac{\pi}{6}$$

$$\omega = \omega_0 = 0$$

We want to know the values $\theta(t)$ and $\omega(t)$ at any given $t > t_0$. We also know the rate at which they are increasing at $t = t_0$:

$$\theta'(t_0) = \omega_0$$

$$\omega'(t_0) = -\sin(\theta_0)$$

Now advance time a little to $t_1 = t_0 + \delta t$, say. By this time θ and ω will roughly change to

$$\theta_1 = \theta_0 + \theta'(t_0)\delta t = \theta_0 + \omega_0 \delta t$$

$$\omega_1 = \omega_0 + \omega'(t_0)\delta t = \omega_0 - \sin(\theta_0)\delta t$$

So we get the phase (approximately) at $t_1 = t_0 + \delta t$. Now we keep on advancing time by δt

increments. The same logic may be used repeatedly to give, at $t_k = t_0 + k \cdot \delta t$,

$$\begin{aligned}\theta_k &= \theta_{k-1} + \omega_{k-1}\delta t\\ \omega_k &= \omega_{k-1} - \sin(\theta_{k-1})\delta t\end{aligned}$$

Admittedly, this is a rather crude approximation. However, if δt is pretty small, the accuracy increases.

(a) Forward Euler:

$$\begin{aligned}\theta_{k+1} &= \theta_k + \omega_k\delta t\\ \omega_{k+1} &= \omega_k - \sin(\theta_k)\delta t\end{aligned}$$

(b) Backward Euler:

$$\begin{aligned}\theta_{k+1} &= \theta_k + \omega_{k+1}\delta t\\ &= \theta_k + \Big(\omega_k - \sin(\theta_k)\delta t\Big)\delta t\\ \omega_{k+1} &= \omega_k - \sin(\theta_{k+1})\delta t\\ &= \omega_k - \sin(\theta_k + \omega_k\delta t)\delta t\end{aligned}$$

(c) Trapezoidal rule:

$$\begin{aligned}\theta_{k+1} &= \theta_k + \frac{\delta t}{2}\Big(\omega_k + \omega_{k+1}\Big)\\ &= \theta_k + \frac{\delta t}{2}\Big(\omega_k + \omega_k - \sin(\theta_k)\delta t\Big)\end{aligned}$$

$$\begin{aligned}\omega_{k+1} &= \omega_k - \frac{\delta t}{2}\Big(\sin(\theta_k) + \sin(\theta_{k+1})\Big)\\ &= \omega_k - \frac{\delta t}{2}\Big(\sin(\theta_k) + \sin(\theta_k + \omega_k\delta t)\Big)\end{aligned}$$

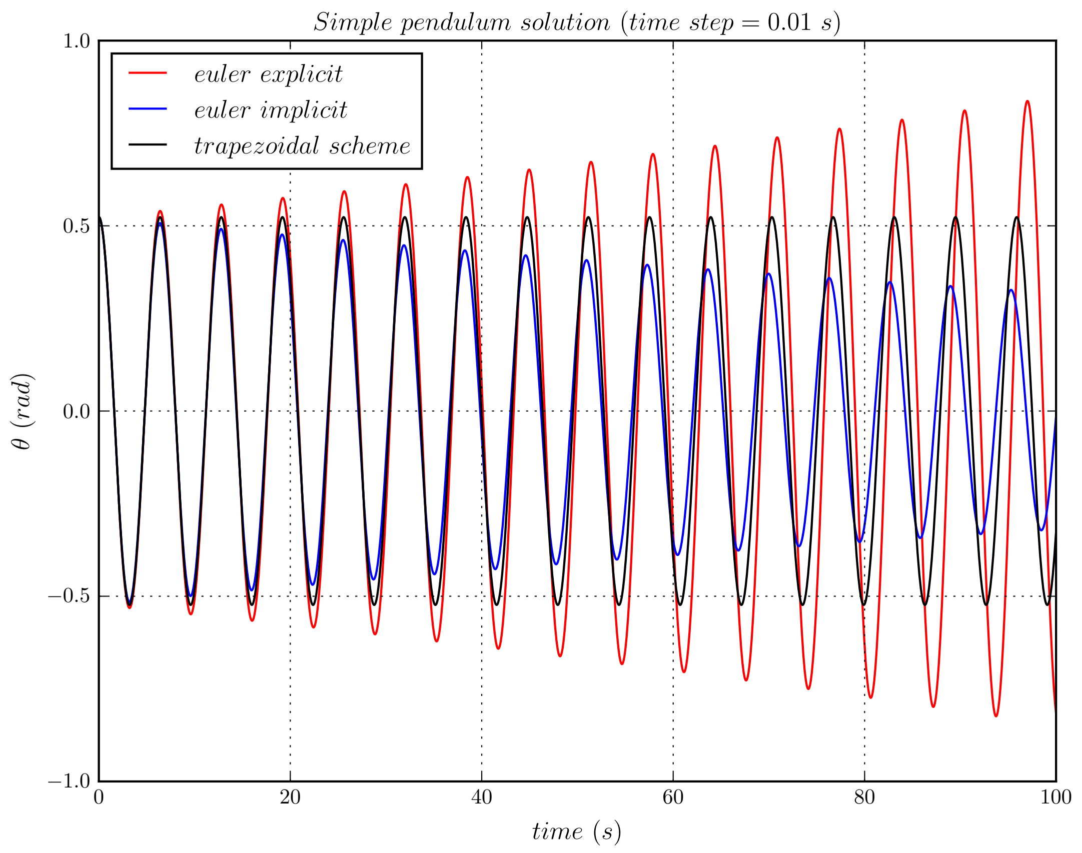
Program:

```
#!/usr/bin/env python
# File: program8.py
# Name: D.Saravanan
# Date: 02/12/2021

""" Script to compute the angular displacement and angular velocity for a simple
pendulum using Euler's method """

```

```python
import matplotlib.pyplot as plt
import numpy as np

plt.style.use("classic")
plt.rcParams["text.usetex"] = True
plt.rcParams["pgf.texsystem"] = "pdflatex"
plt.rcParams.update(
    {
        "font.family": "serif",
        "font.size": 10,
        "axes.labelsize": 12,
        "axes.titlesize": 12,
        "figure.titlesize": 12,
    }
)

t_ = 100  # time period (s)

# time step (s)
for n, dt in enumerate([0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20]):

    N = int(t_ / dt)  # number of steps
    t = np.zeros(N + 1)  # time vector (s)

    theta1 = np.ones(N + 1)  # angular displacement
    omega1 = np.ones(N + 1)  # angular velocity

    theta1[0] = np.pi / 6  # initial value of angular displacement (rad)
    omega1[0] = 0.0  # initial value of angular velocity (1/s)

    theta2 = np.ones(N + 1)  # angular displacement
    omega2 = np.ones(N + 1)  # angular velocity

    theta2[0] = np.pi / 6  # initial value of angular displacement (rad)
    omega2[0] = 0.0  # initial value of angular velocity (1/s)

    theta3 = np.ones(N + 1)  # angular displacement
    omega3 = np.ones(N + 1)  # angular velocity

    theta3[0] = np.pi / 6  # initial value of angular displacement (rad)
    omega3[0] = 0.0  # initial value of angular velocity (1/s)

    for k in range(0, N):

        t[k + 1] = t[k] + dt

        # forward euler
        theta1[k + 1] = theta1[k] + omega1[k] * dt
        omega1[k + 1] = omega1[k] - np.sin(theta1[k]) * dt

        # backward euler
        theta2[k + 1] = theta2[k] + (omega2[k] - np.sin(theta2[k]) * dt) * dt
        omega2[k + 1] = omega2[k] - np.sin(theta2[k] + omega2[k] * dt) * dt

        # trapezoidal scheme
        theta3[k + 1] = theta3[k] + (dt / 2) * (
            omega3[k] + omega3[k] - np.sin(theta3[k]) * dt
        )
        omega3[k + 1] = omega3[k] - (dt / 2) * (
            np.sin(theta3[k]) + np.sin(theta3[k] + dt * omega3[k])
        )

    figure, ax = plt.subplots()
    ax.plot(t, theta1, "r--", label=r"$euler\ explicit$")
    ax.plot(t, theta2, "b--", label=r"$euler\ implicit$")
    ax.plot(t, theta3, "k--", label=r"$trapezoidal\ scheme$")
    ax.set(xlabel=r"$time\ (s)$", ylabel=r"$\theta\ (rad)$")
    ax.set_title(r"$Simple\ pendulum\ solution\ (time\ step = {}\ s)$".format(dt))
    ax.grid(True); ax.legend(loc="best")
    plt.savefig("program8{}.pgf".format(n))
```

From the plots,

- Forward Euler scheme become unstable at time step, $\delta t = 0.05s$
- Backward Euler scheme become unstable at time step, $\delta t = 2s$
- Trapezoidal Rule scheme become unstable at time step, $\delta t = 1s$



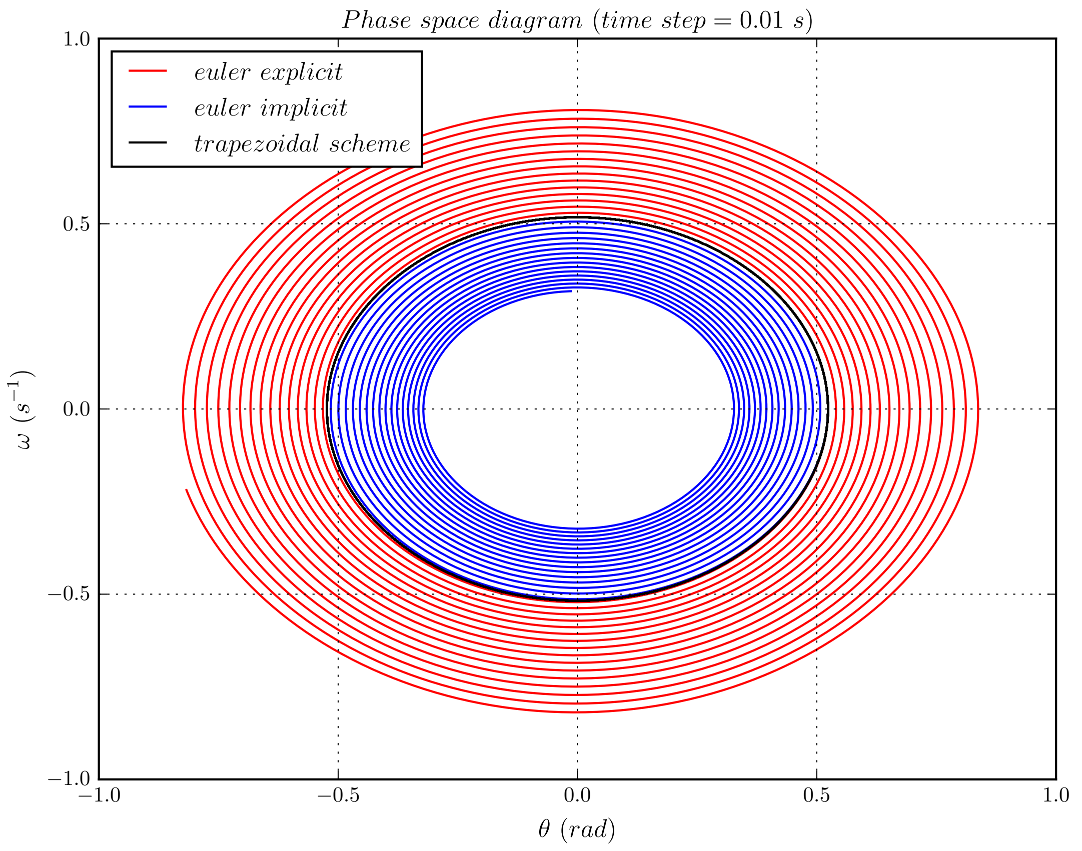
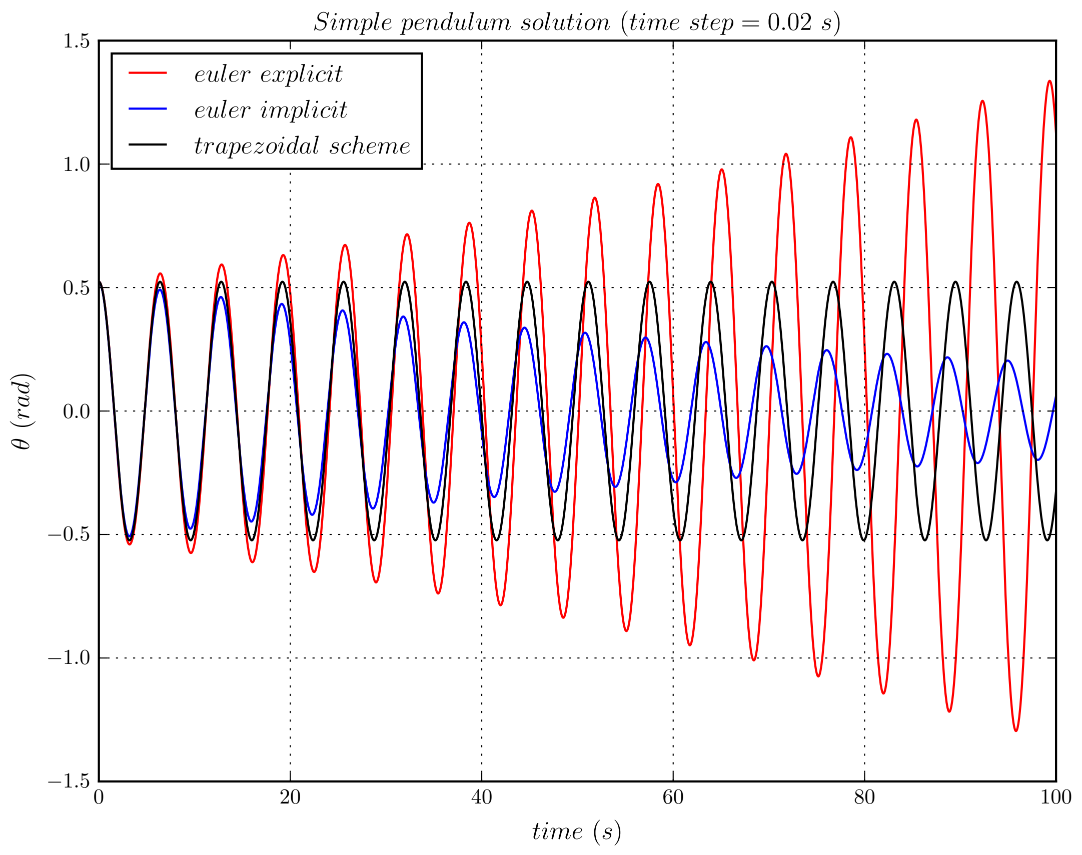


Figure 3: Numerical results for the nonlinear simple pendulum equation with $\delta t = 0.01s$



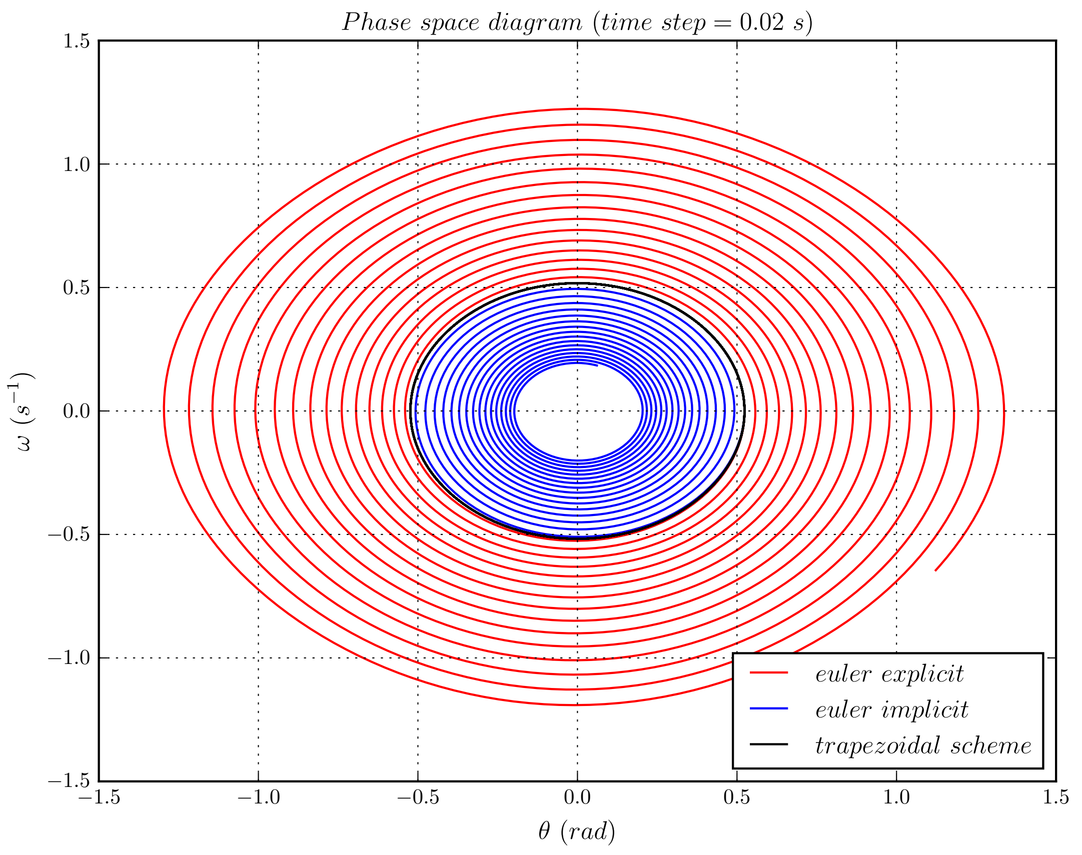
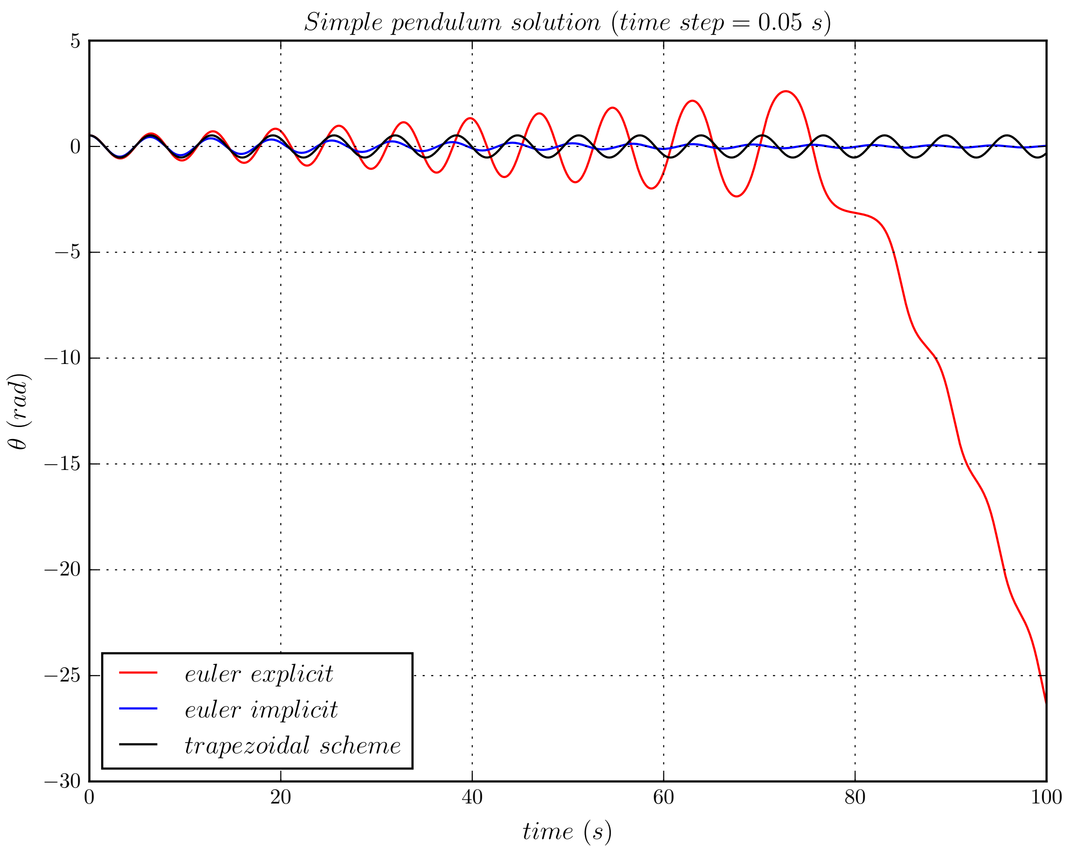


Figure 4: Numerical results for the nonlinear simple pendulum equation with $\delta t = 0.02s$



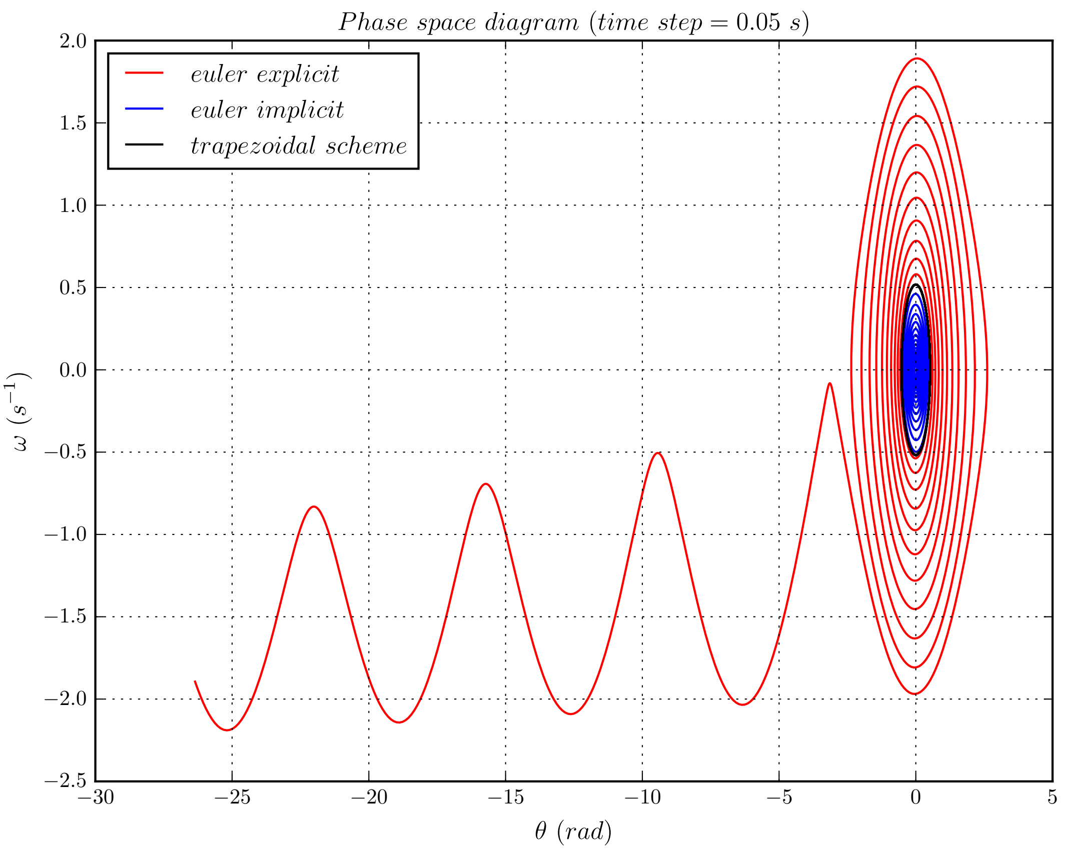
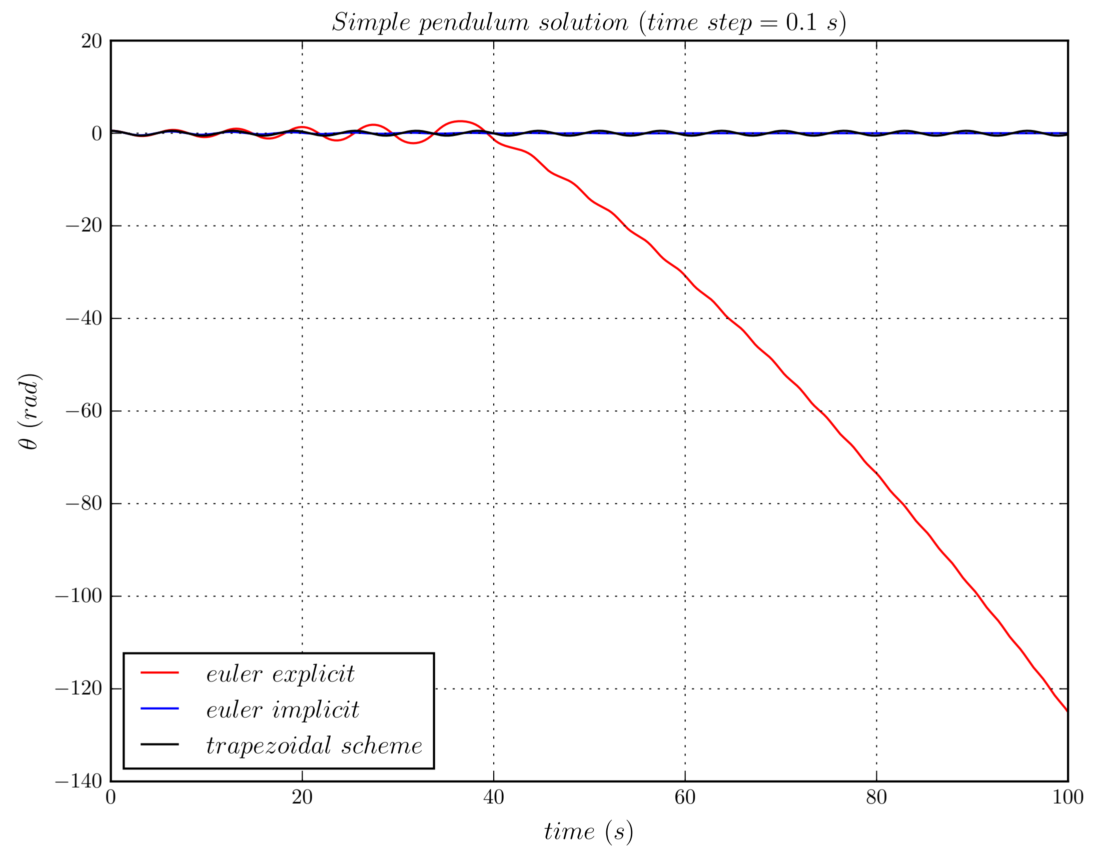


Figure 5: Numerical results for the nonlinear simple pendulum equation with $\delta t = 0.05s$



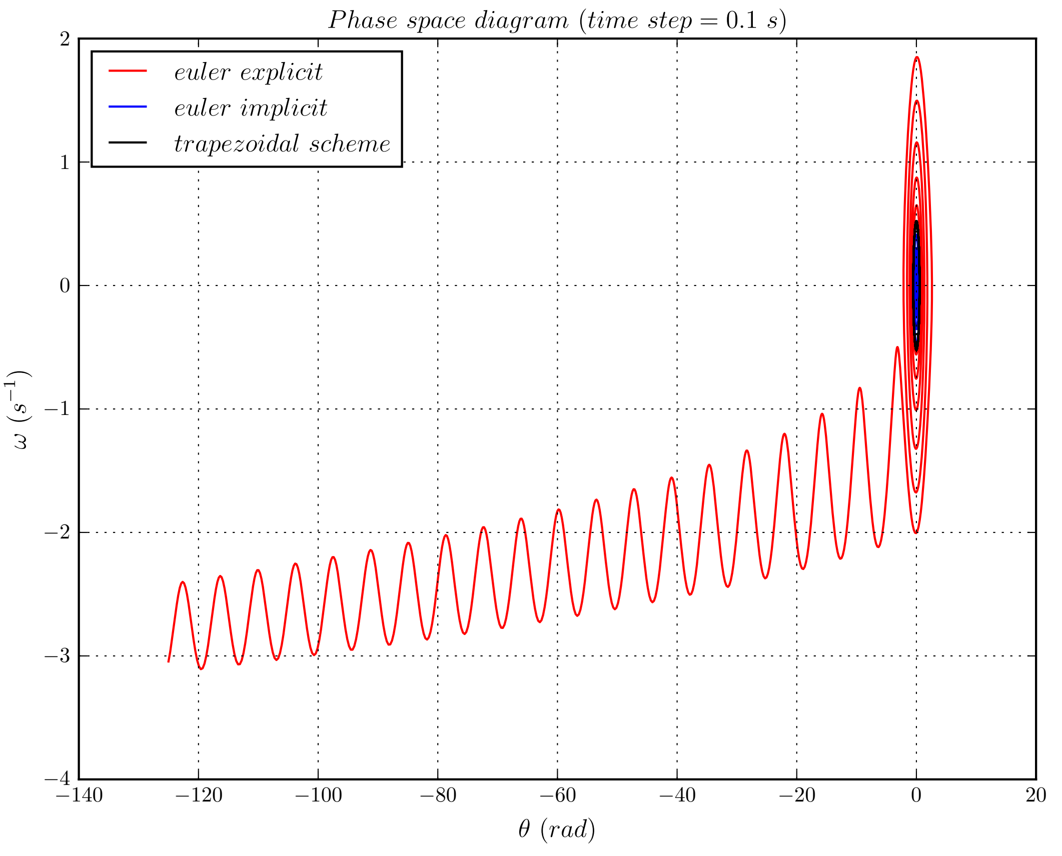
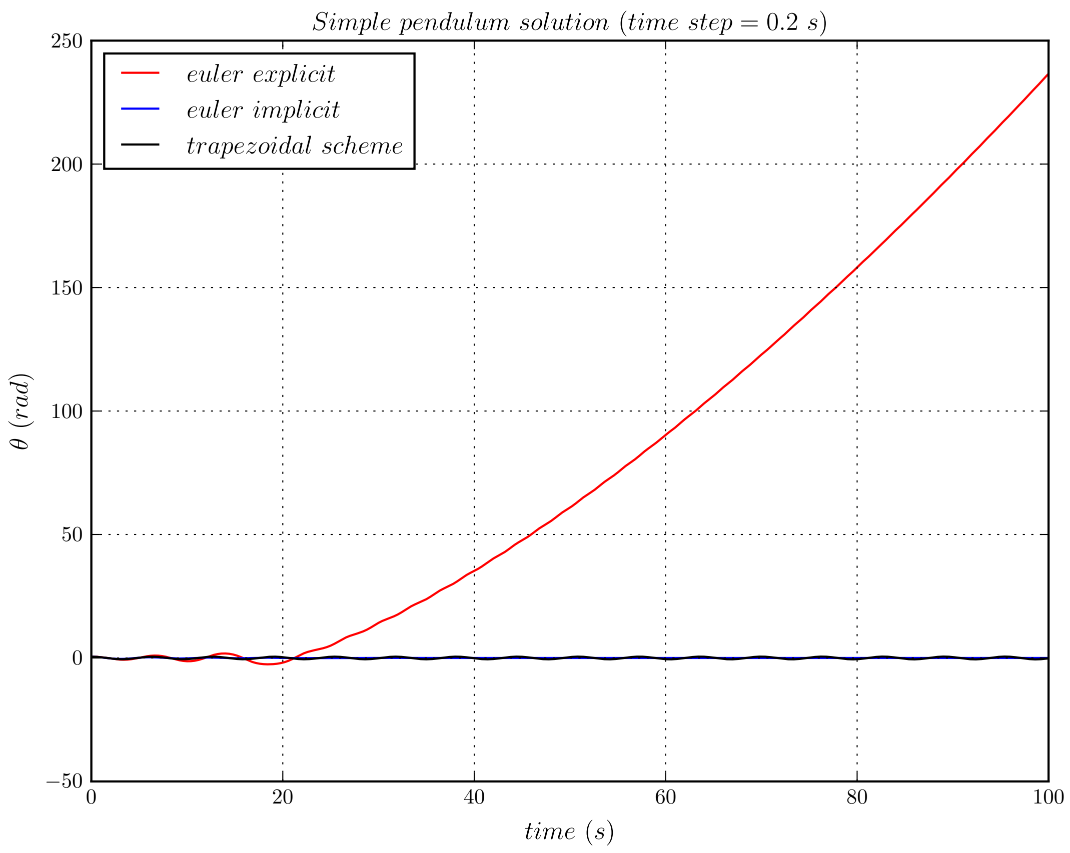


Figure 6: Numerical results for the nonlinear simple pendulum equation with $\delta t = 0.1s$



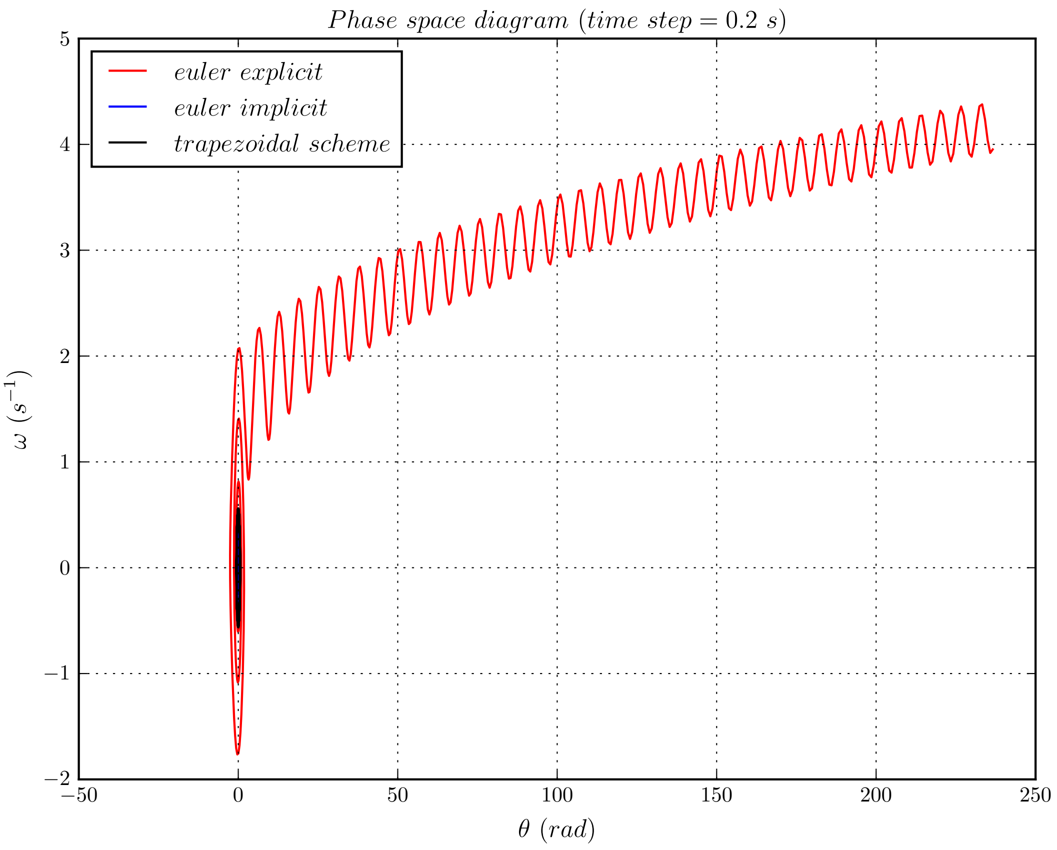
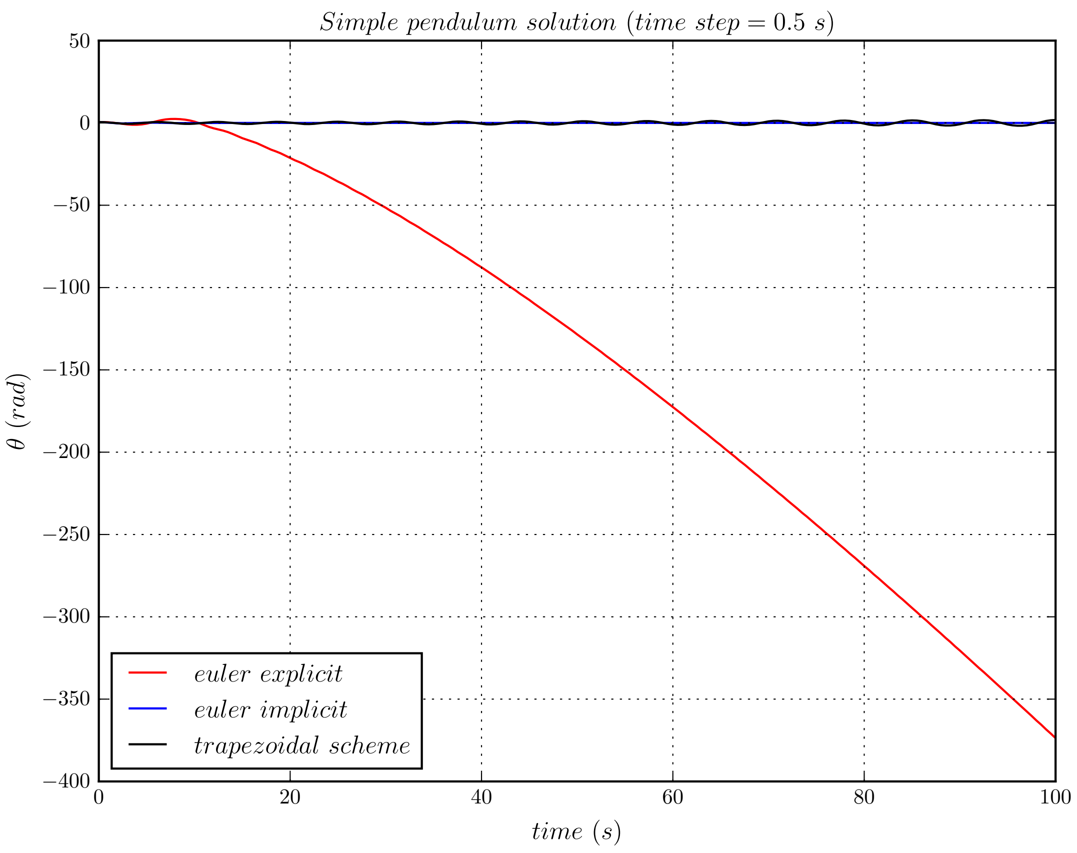


Figure 7: Numerical results for the nonlinear simple pendulum equation with $\delta t = 0.2s$



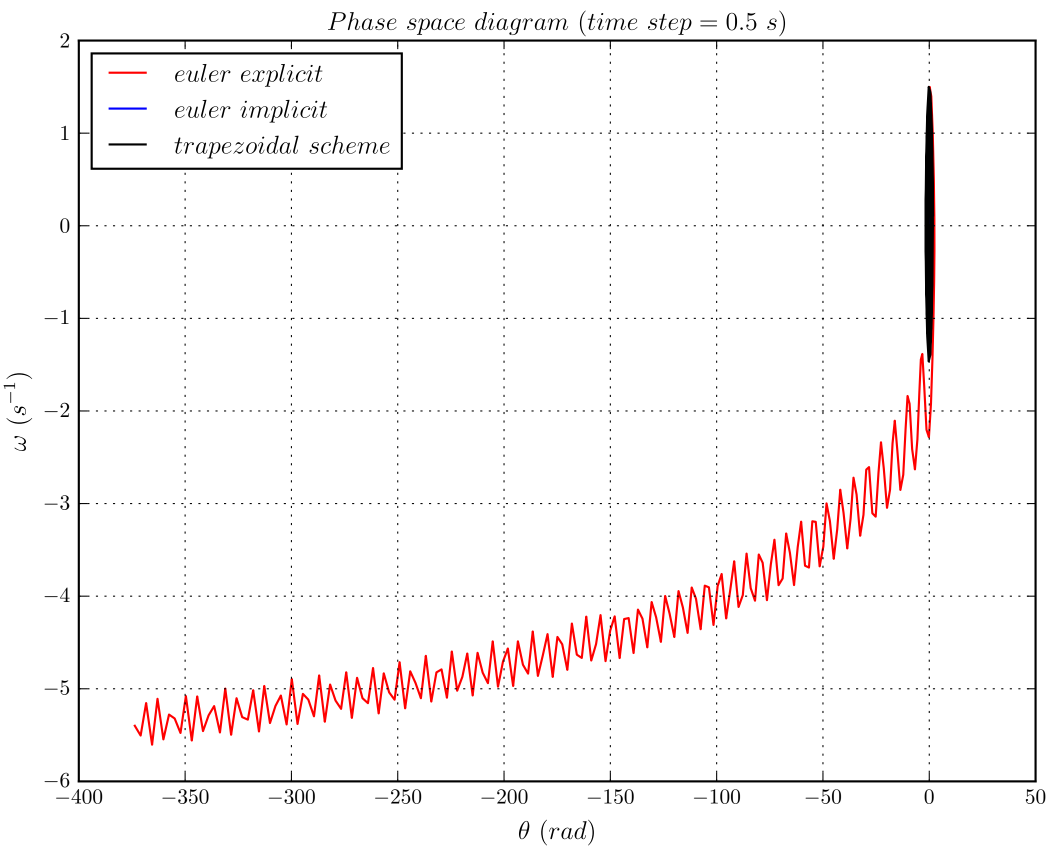
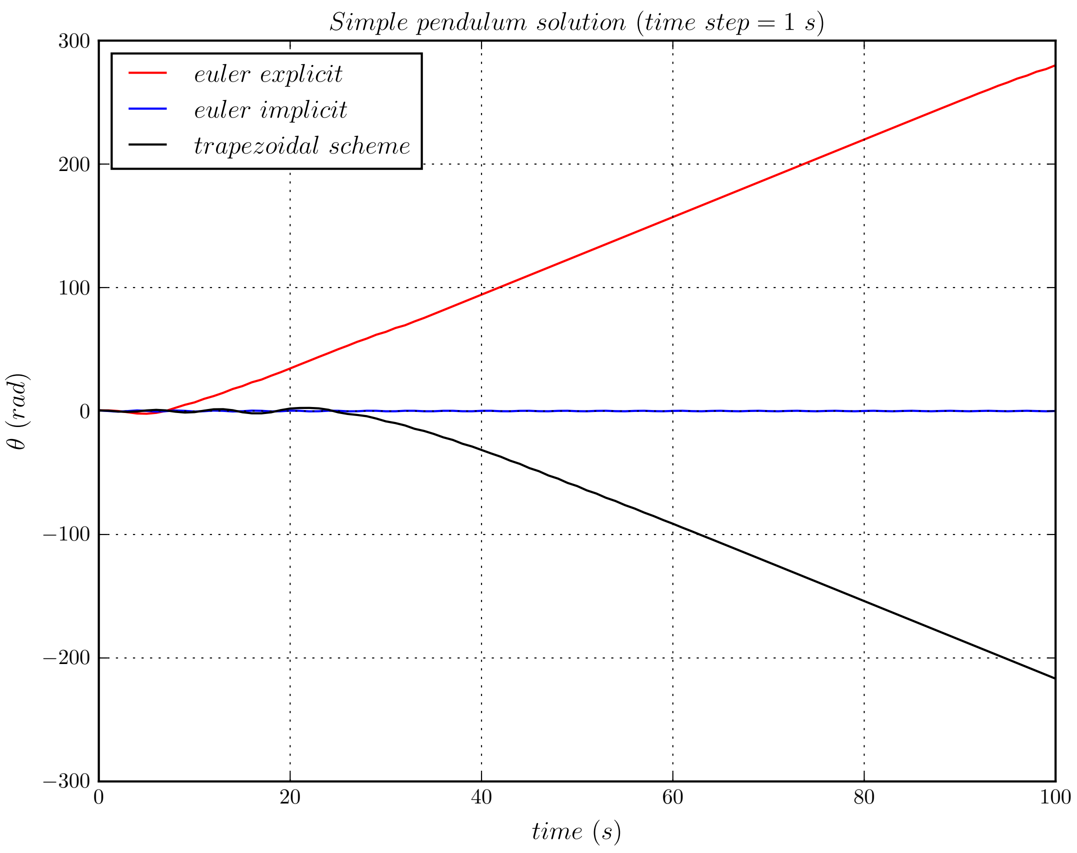


Figure 8: Numerical results for the nonlinear simple pendulum equation with $\delta t = 0.5s$



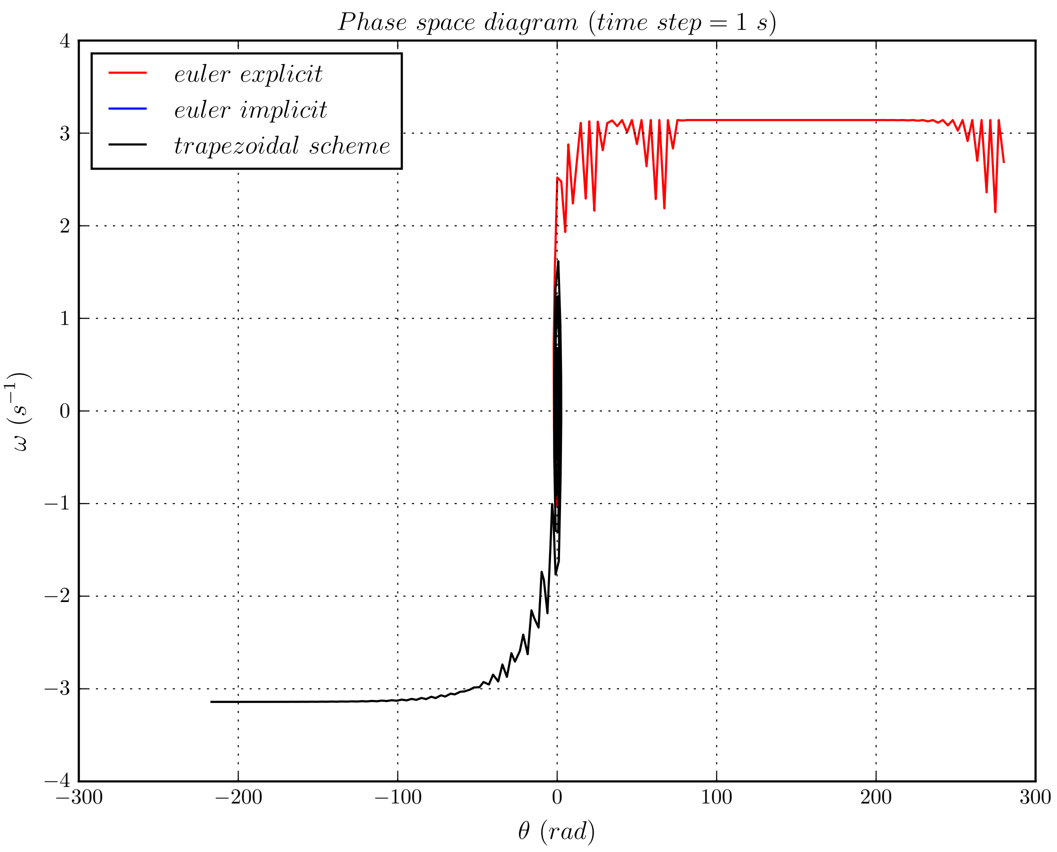
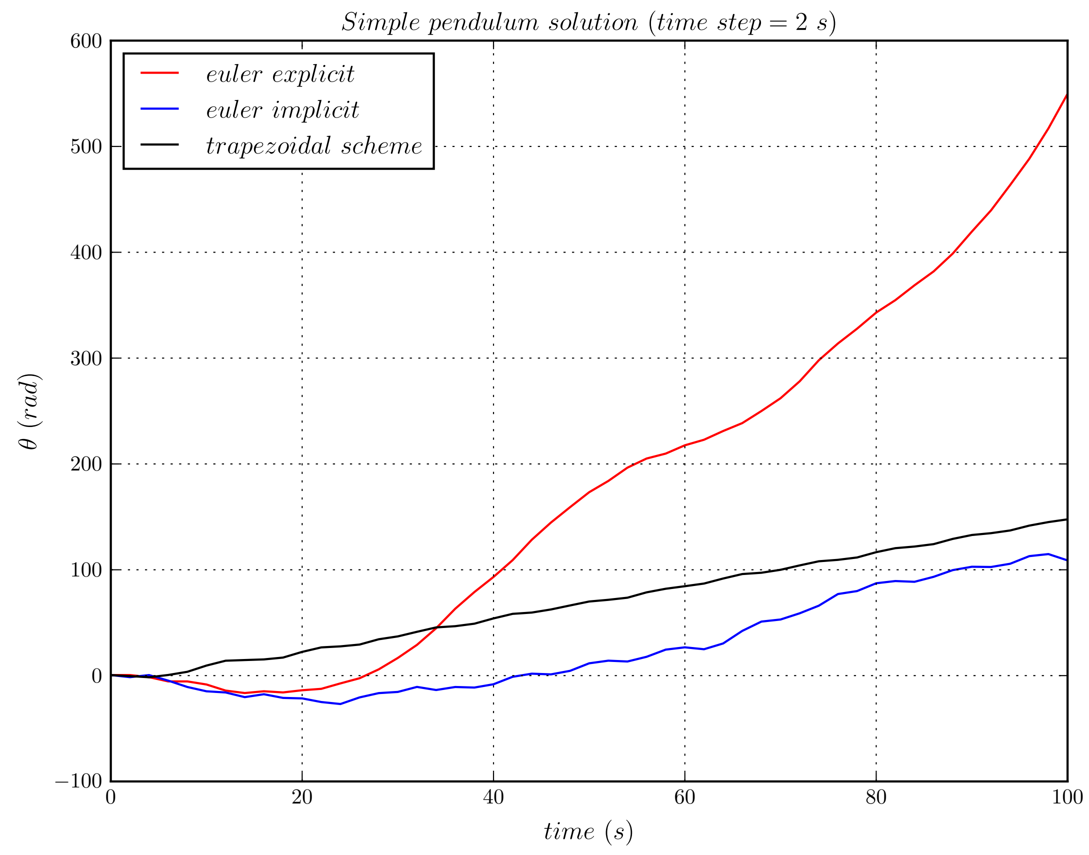


Figure 9: Numerical results for the nonlinear simple pendulum equation with $\delta t = 1s$



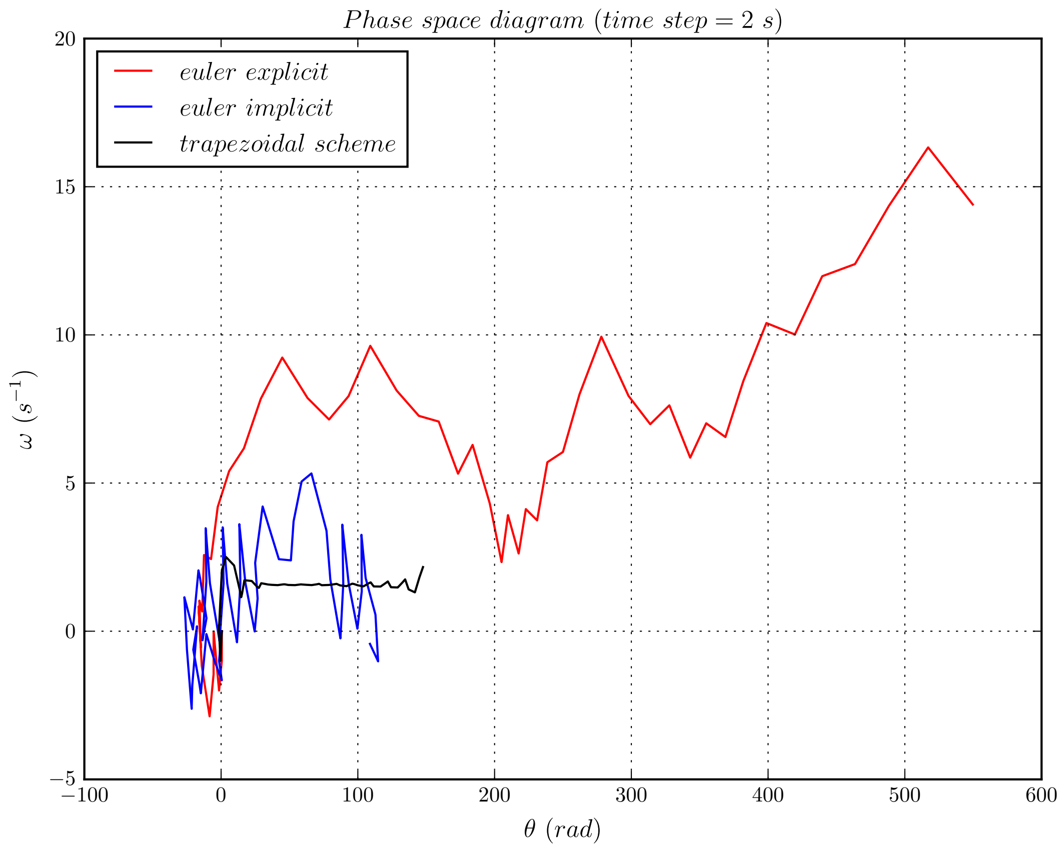
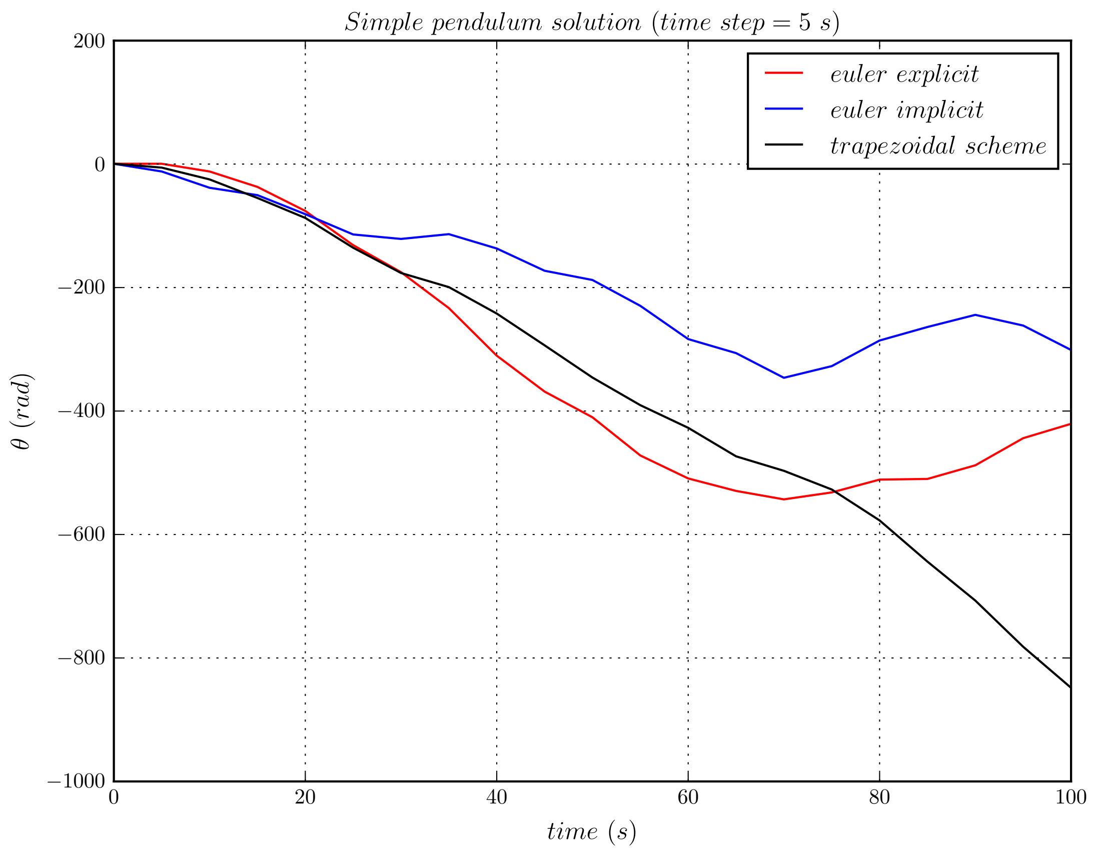


Figure 10: Numerical results for the nonlinear simple pendulum equation with $\delta t = 2s$



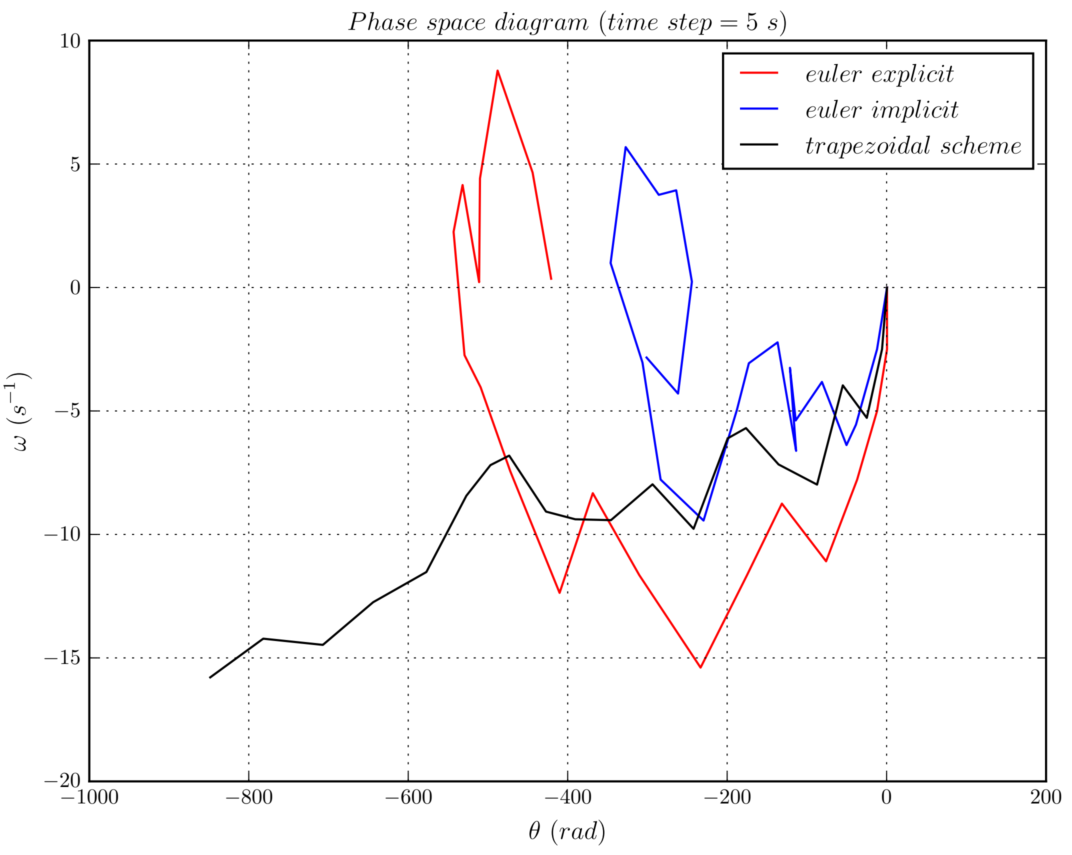
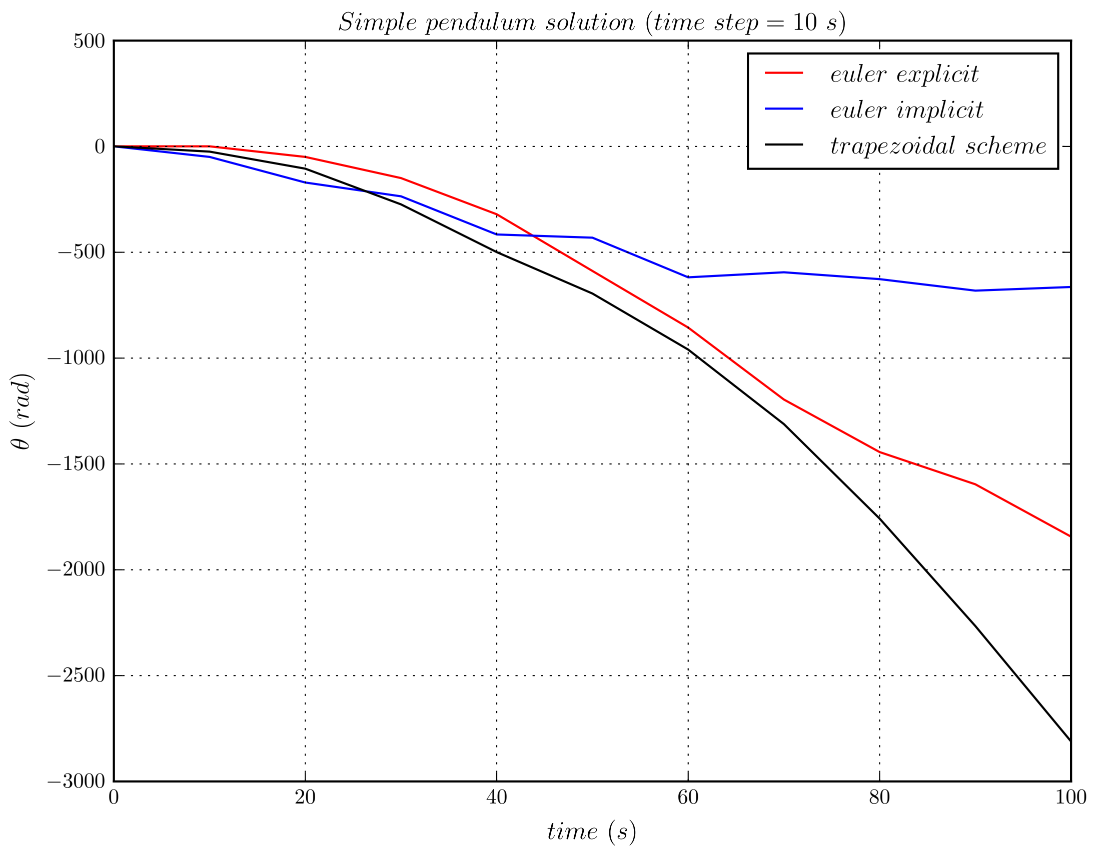


Figure 11: Numerical results for the nonlinear simple pendulum equation with $\delta t = 5s$



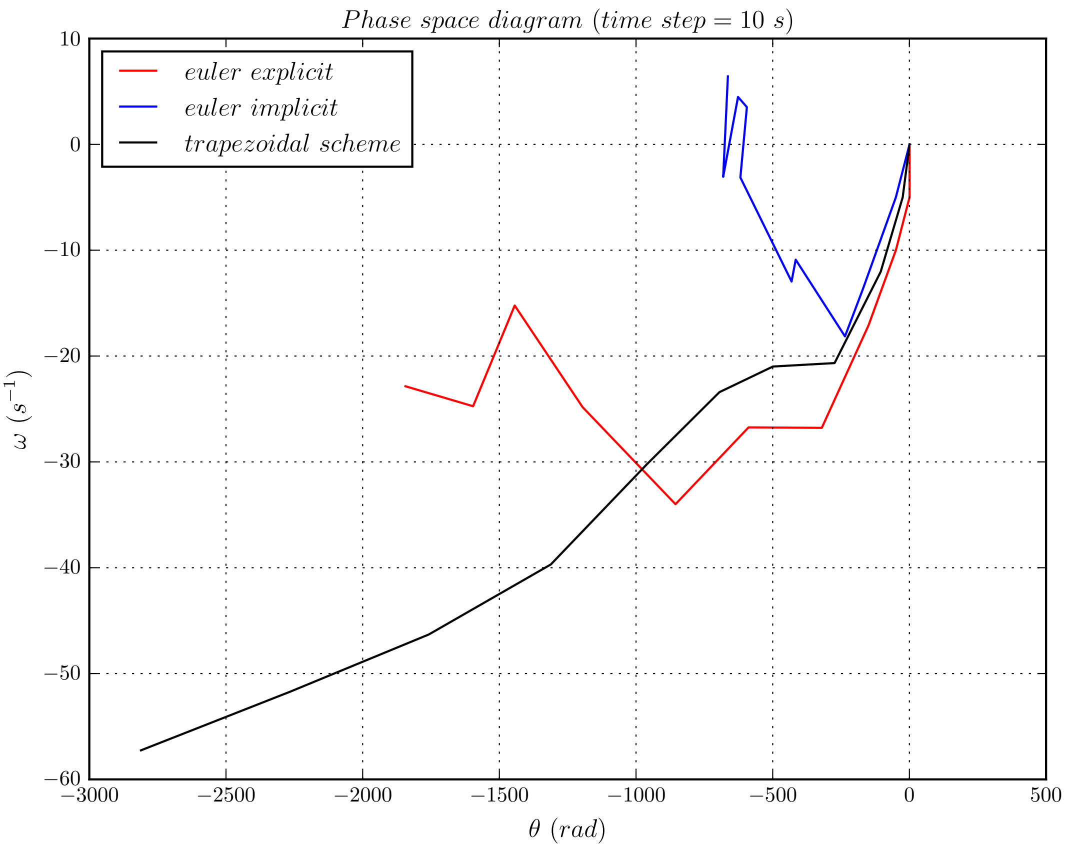
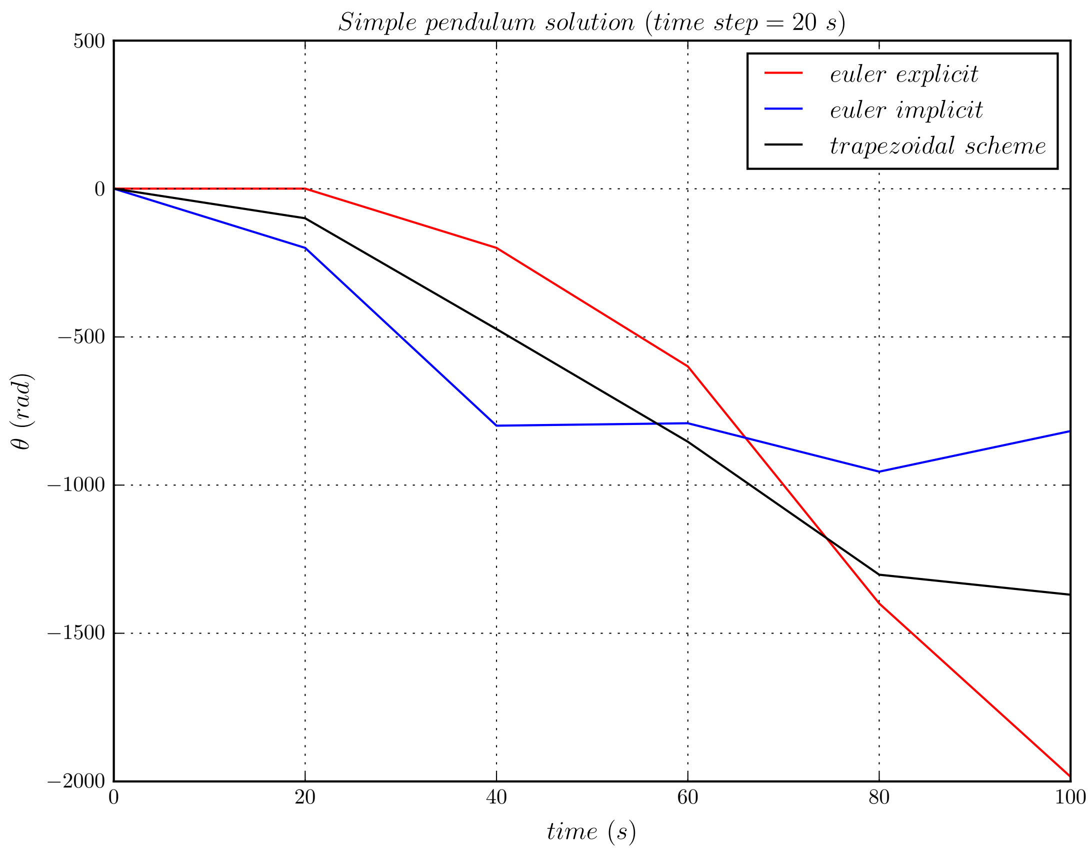


Figure 12: Numerical results for the nonlinear simple pendulum equation with $\delta t = 10s$



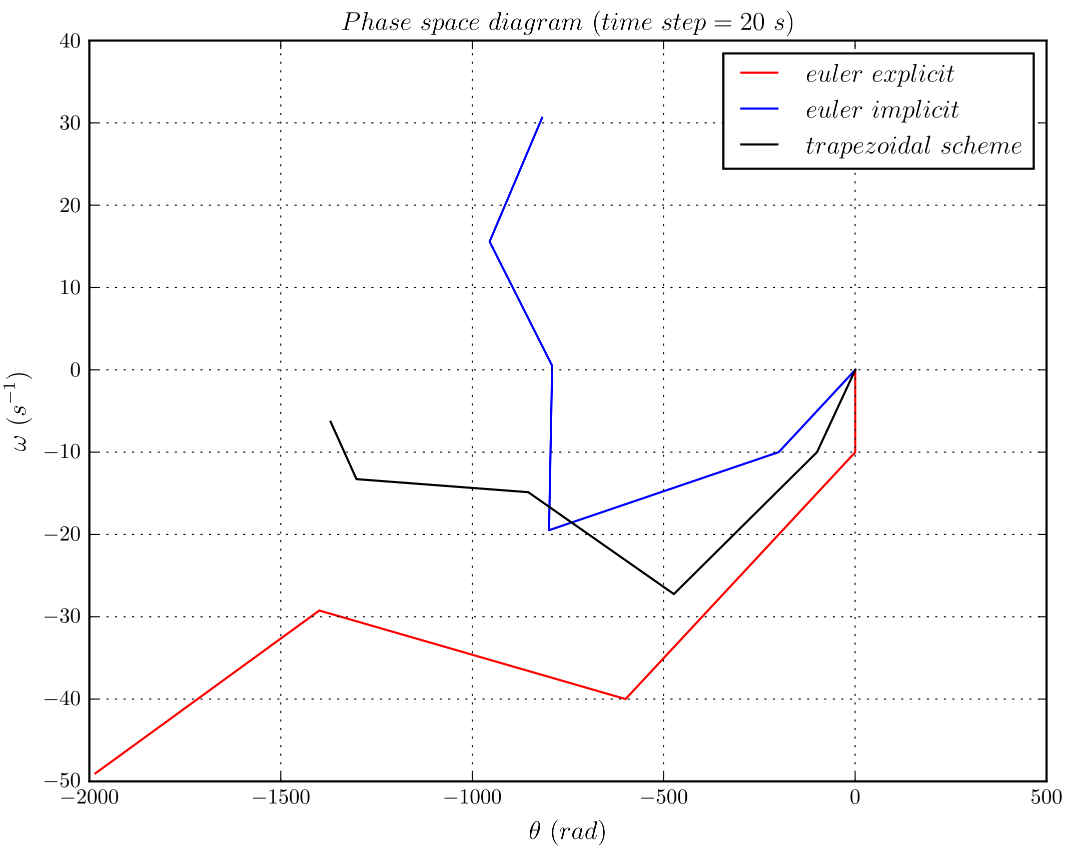


Figure 13: Numerical results for the nonlinear simple pendulum equation with $\delta t = 20s$

Physical stability for the simple pendulum:

The stability analysis of the simple pendulum problem will be done by approximating $sin(\theta)$ to be θ.

$$\frac{d^2\theta}{dt^2} = -\theta$$

converting into a system of first order ordinary differential equations,

$$\theta' = \frac{d\theta}{dt} = \omega$$

$$\theta'' = \frac{d\omega}{dt} = -\theta$$

rewriting in matrix form,

$$\frac{d}{dt}\begin{bmatrix} \theta \\ \omega \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \omega \end{bmatrix}$$
$$\frac{d}{dt}S(t) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} S(t) = F(S,t)$$

then,

$$\det\left(A - \lambda I\right) = 0$$

$$\det\left(\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = 0$$
$$\det\left(\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix}\right) = 0$$
$$\det\left(\begin{bmatrix} -\lambda & 1 \\ -1 & -\lambda \end{bmatrix}\right) = 0$$

solving, we obtain $\lambda = \pm i$.

For the exact solution to be stable and bounded, the $Re(\lambda) \leq 0$. Since, $Re(\lambda) = 0$, the exact solution for the simple pendulum problem is stable and bounded.

- Forward Euler:

$$\begin{aligned}S_{n+1} &= S_n + \delta t\ F(S_n, t_n)\\ &= S_n + \delta t \begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} S_n\\ &= \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} S_n + \delta t \begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} S_n\end{aligned}$$

$$S_{n+1} = \begin{bmatrix}1 & \delta t\\ -\delta t & 1\end{bmatrix} S_n$$

- Backward Euler:

$$\begin{aligned}S_{n+1} &= S_n + \delta t\ F(S_{n+1}, t_{n+1})\\ &= S_n + \delta t \begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} S_{n+1}\end{aligned}$$

$$\begin{aligned}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} S_{n+1} - \delta t \begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} S_{n+1} &= S_n\\ \begin{bmatrix}1 & -\delta t\\ \delta t & 1\end{bmatrix} S_{n+1} &= S_n\end{aligned}$$

$$S_{n+1} = \begin{bmatrix}1 & -\delta t\\ \delta t & 1\end{bmatrix}^{-1} S_n$$

- Trapezoidal rule:

$$\begin{aligned}S_{n+1} &= S_n + \frac{\delta t}{2}\left[F(S_n, t_n) + F(S_{n+1}, t_{n+1})\right]\\ &= S_n + \frac{\delta t}{2}\left(\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} S_n + \begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} S_{n+1}\right)\end{aligned}$$

$$\begin{aligned}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} S_{n+1} - \frac{\delta t}{2}\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} S_{n+1} &= \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} S_n + \frac{\delta t}{2}\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} S_n\\ \left(\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} - \begin{bmatrix}0 & \frac{\delta t}{2}\\ -\frac{\delta t}{2} & 0\end{bmatrix}\right) S_{n+1} &= \left(\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} + \begin{bmatrix}0 & \frac{\delta t}{2}\\ -\frac{\delta t}{2} & 0\end{bmatrix}\right) S_n\end{aligned}$$

$$\begin{bmatrix} 1 & -\frac{\delta t}{2} \\ \frac{\delta t}{2} & 1 \end{bmatrix} S_{n+1} = \begin{bmatrix} 1 & \frac{\delta t}{2} \\ -\frac{\delta t}{2} & 1 \end{bmatrix} S_n$$

$$S_{n+1} = \begin{bmatrix} 1 & -\frac{\delta t}{2} \\ \frac{\delta t}{2} & 0 \end{bmatrix}^{-1} \begin{bmatrix} 1 & \frac{\delta t}{2} \\ -\frac{\delta t}{2} & 1 \end{bmatrix} S_n$$

These equations allow us to solve the initial value problem since at each state, S_n, we can compute the next state at S_{n+1}. In general, this is possible to do when an ODE is linear.