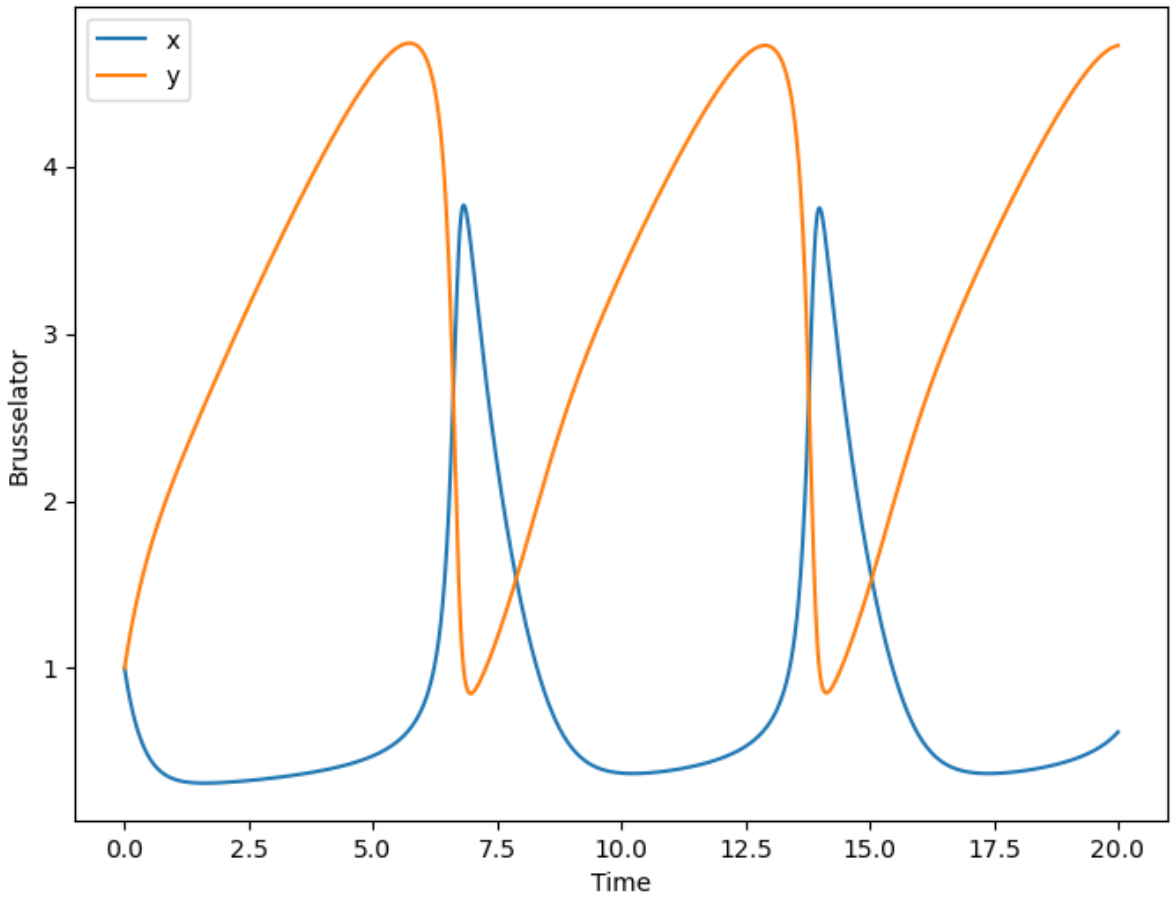
Scientific Computing Report

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp
import scipy.linalg as LA
import timeit
```

Question 1: To solve the `Brusselator` system across a specific time interval, (0,20), `solve_ode` from `ode_solver` is imported, along with the system of equations defined in `Equations_Functions`, which is a python file which contains systems of equations used in both the demos and testing, for the users understanding I have put the system of equations in the demo. The `solve_ode` function has two step methods, `rk4_step` and `euler_step`. It takes an initial guess and an array t, for which the system will be evaluated for. Parameters are passed in at the end, assure the parameters are in the correct order - this depends on how the system is defined in Equations_Functions. In this case it is [B, A]. This approach is to be repeated in question 2, therefore I will only mention this once.

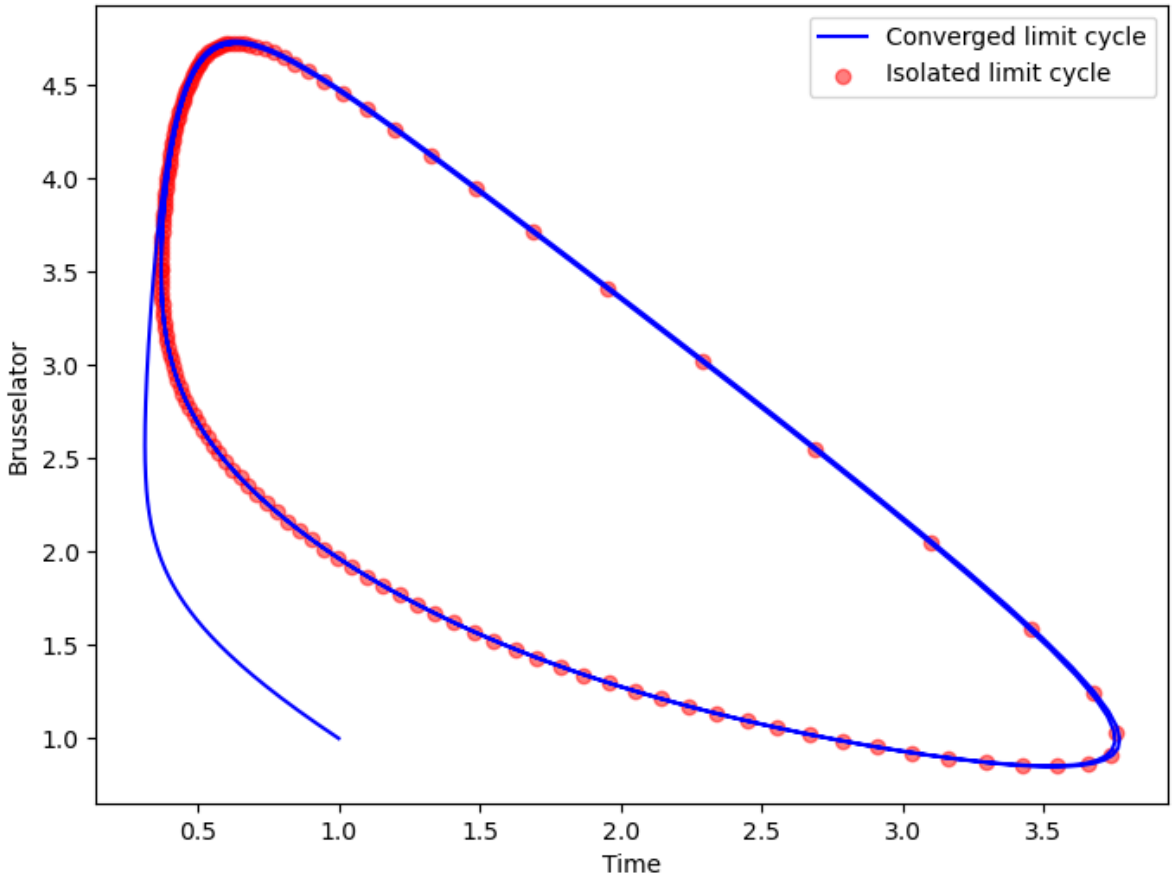
In [ ]:
```
#a)
#from Equations_Functions import brusselator
from ode_solver import solve_ode

def brusselator(t,u,pars):
    A = pars[1]
    B = pars[0]
    u1,u2 = u
    du1dt = A + u1**2 * u2 - (B + 1) * u1
    du2dt = B * u1 - u1**2 * u2

    dUdt = np.array([du1dt,du2dt])
    return dUdt

pars = [3, 1] # B (A is fixed = 1)
y0 = np.array([1, 1])
T = 20
t = np.linspace(0, T, 1000)
sol = solve_ode(brusselator, y0, t, "rk4", 0.05, pars)
plt.figure(figsize=(8, 6))
plt.plot(t, sol[0, :], label='x')
plt.plot(t, sol[1, :], label='y')
plt.xlabel('Time')
plt.ylabel('Brusselator')
plt.legend()
plt.show()
```



To determine the coordinates of a starting point of the limit cycle in part a, we import `shoot` and `limit_cycle_finder`. A suitable phase condition is defined already, the limit cycle finder uses shooting and `scipy.optimize.fsolve` to converge to a limit cycle. The test kwarg is for testing, it notifies if the root finder doesn't converge. This is not needed, hence its optional. The `orbit` function is imported to solve the function using the isolated limit cycle coordinates found by `limit_cycle_finder`, this traces the limit cycle, to then be plotted. As we can see, the red dots are located on the converging limit cycle from part a. This illustrates that limit cycle finder produces the correct results.

In [ ]:
```
from scipy.optimize import fsolve
from bvp_and_shooting import shoot, limit_cycle_finder, orbit

def phase_condition(ode,u0,pars):
    return np.array([ode(0,u0,pars)[0]])

pars = [3,1] #B
y0 = np.array([2,3,7])
sol_lc = limit_cycle_finder(brusselator, y0,phase_condition,pars, descretiza
cycle, t = orbit(brusselator, sol_lc[:-1], sol_lc[-1], pars)
print(f'The starting point: {", ".join([f"{val:.2f}" for val in sol_lc[:-1]]

plt.figure(figsize=(8, 6))
plt.plot(sol[0, :],sol[1,:], label='Converged limit cycle', c='blue')
plt.scatter(cycle[0, :],cycle[1,:], label='Isolated limit cycle', c='red', a
plt.xlabel('Time')
plt.ylabel('Brusselator')
plt.legend()
plt.show()
```

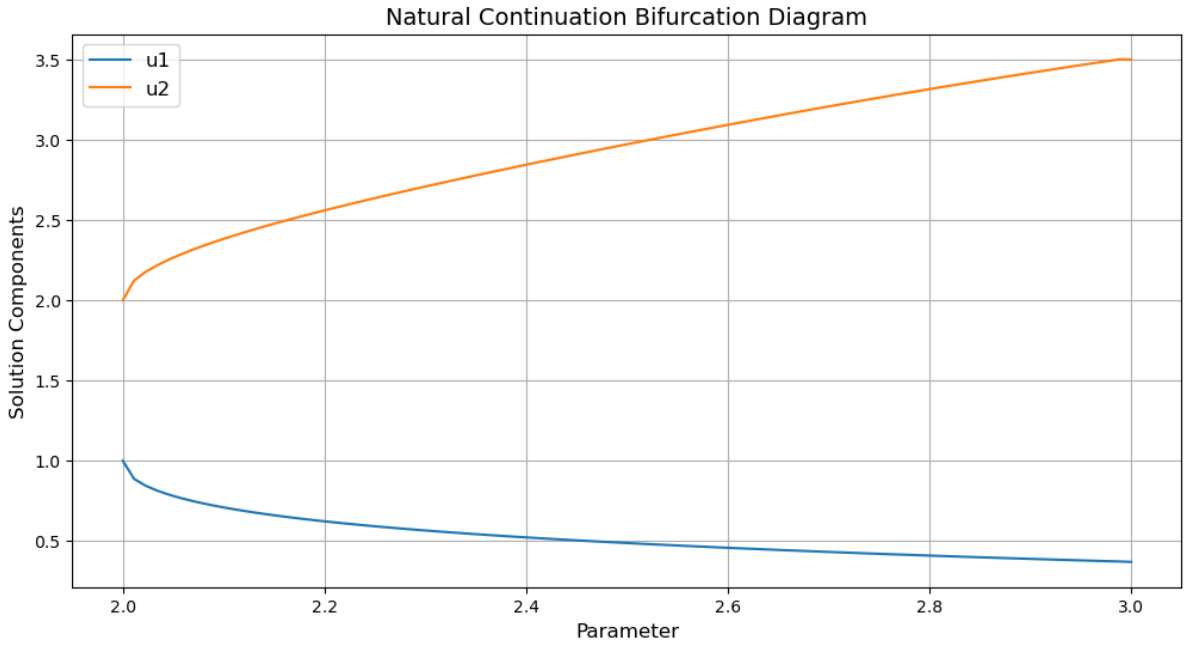
```
The starting point: 0.37, 3.51 values and period: 7.16 for the Brusselator
orbit
```



To naturally continue over the range 2 < B < 3, `numerical_continuation` can be imported from `numerical_continuation`. To choose which parameter to continue over, use par_index in correspondence to the parameters in par_array. In this case we set par_index to 0 to continue over B, should be noted that the value of the paremter in par_array doesnt matter for natural continutation because it continues from the max param value to min param value. For pseudo arc length continuation, the value in par_array is the starting value so this must match the max par in par bounds. To use natural continuation, we use 'natural' in the method argument, the other method which can be used here is 'pseudo' if pseudo arc-length continuation is the desired method. When using `natural plotter`, the period kwarg is True if we want to track limit cycles (essentially if phase condition = None), this ensures the plotter plots the state variables correctly.

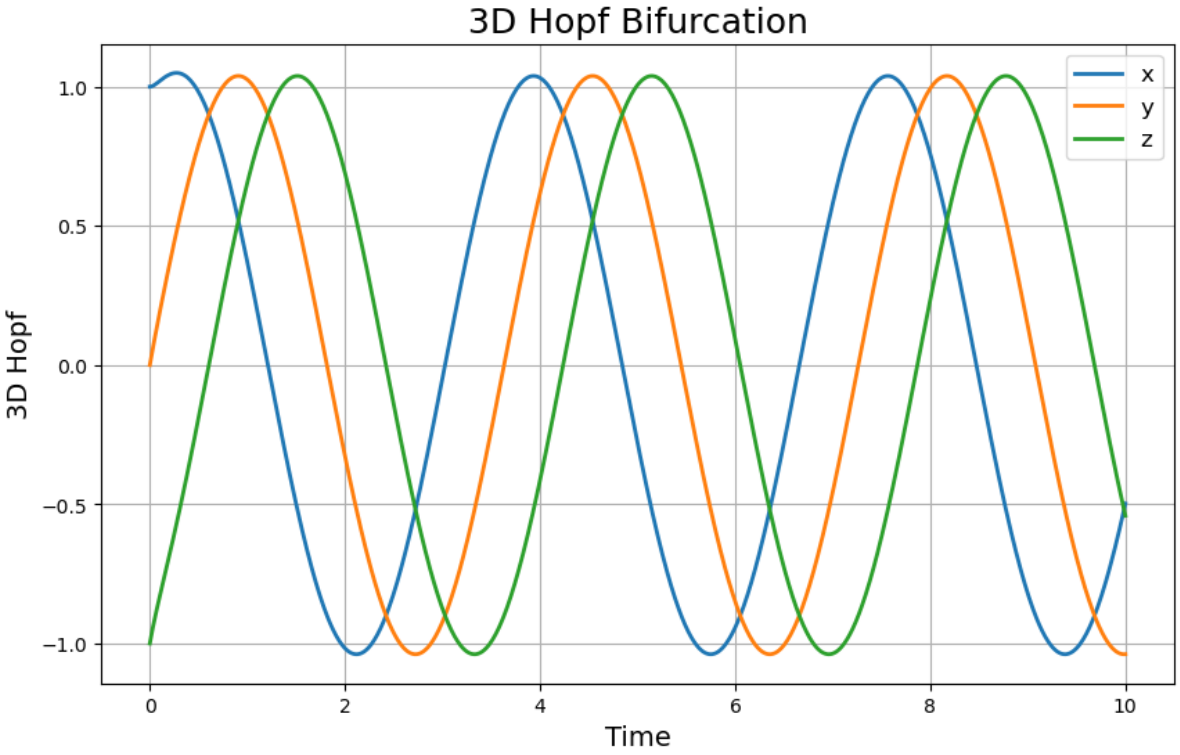
In [ ]:
```
# c)
from numerical_continuation import numerical_continuation, natural_plotter

x0 = np.array([0.37, 3.5, 7.2])
par_array = [3,1]  #[B,A]
par_index = 0

par_nat, sol_nat = numerical_continuation(brusselator,
    'natural',
    x0,
    par_array,
    par_index,
    [3, 2], #Bounds [Max,Min]
```

```
    [200, 90], #Max steps: [PAL, Natural]
    shoot, #discretization
    fsolve, #solver
    phase_condition=phase_condition,
    increase=False) #increase parameter


#Bifurcation diagram of Limit cycle and equilibria
natural_plotter(par_nat, sol_nat, period=True)
```



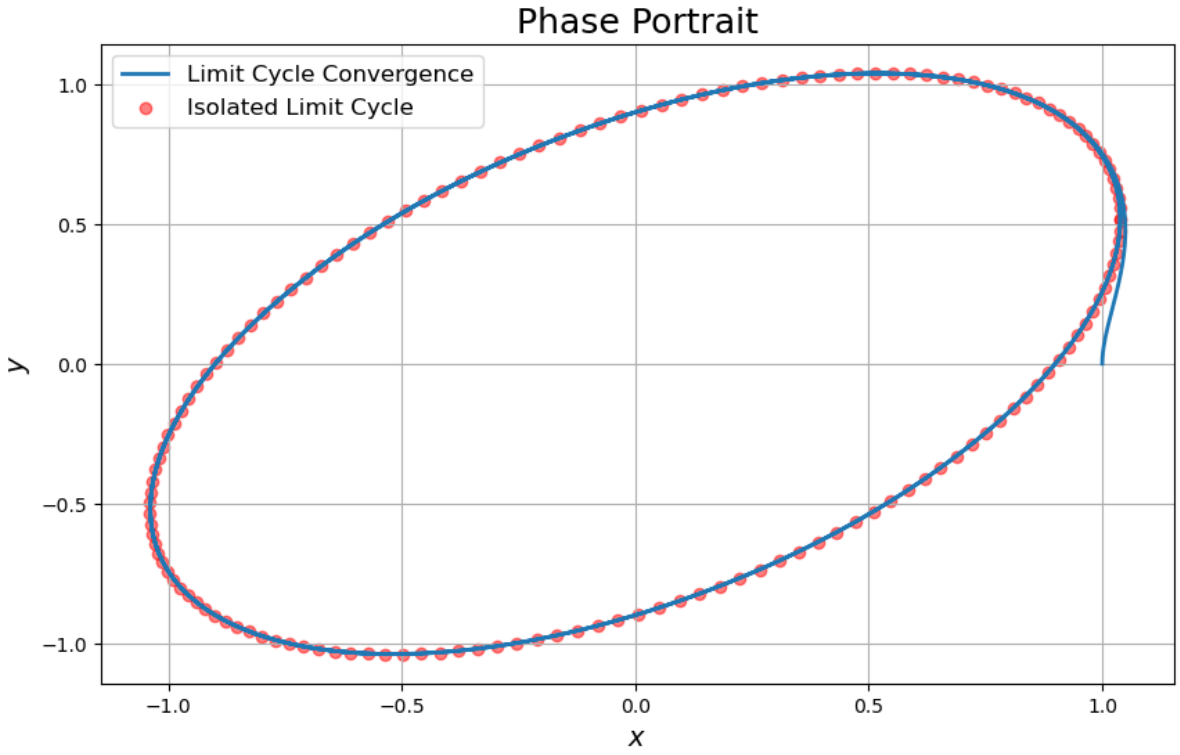
In [ ]:

```
#Question 2:
from Equations_Functions import hopf_bifurcation_3d
def hopf_bifurcation_3d(t, X, pars):
    x, y, z = X
    beta = pars[0]
    r_squared = x**2 + y**2 + z**2  # Common term (r^2)

    dxdt = beta*x - y - z + x*r_squared - x*r_squared**2
    dydt = x + beta*y - z + y*r_squared - y*r_squared**2
    dzdt = x + y + beta*z + z*r_squared - z*r_squared**2

    return [dxdt, dydt, dzdt]

pars = [1] # beta
y0 = np.array([1, 0, -1])
T = 10
t = np.linspace(0, T, 1000)
sol = solve_ode(hopf_bifurcation_3d, y0, t, "rk4", 0.01, pars)

plt.figure(figsize=(10, 6))
plt.plot(t, sol[0, :], label='x', linewidth=2)
plt.plot(t, sol[1, :], label='y', linewidth=2)
plt.plot(t, sol[2, :], label='z', linewidth=2)
plt.xlabel('Time', fontsize=14)
plt.ylabel('3D Hopf', fontsize=14)
plt.title('3D Hopf Bifurcation', fontsize=18)
plt.legend(fontsize=12)
plt.grid(True)
plt.show()
```



In [ ]:
```
# b)

from bvp_and_shooting import phase_condition, shoot, limit_cycle_finder, orb

pars = [1] # beta
y0 = np.array([1, 0, -1, 4])
sol_lm = limit_cycle_finder(hopf_bifurcation_3d, y0,phase_condition,pars,tes
cycle, t = orbit(hopf_bifurcation_3d, sol_lm[:-1], sol_lm[-1], pars)
print(f'The starting point: {", ".join([f"{val:.2f}" for val in sol_lm[:-1]]

plt.figure(figsize=(10, 6))
plt.plot(sol[0, :], sol[1, :], label='Limit Cycle Convergence', linewidth=2,
plt.scatter(cycle[0, :], cycle[1, :], label='Isolated Limit Cycle', c='red',
plt.xlabel('$x$', fontsize=14)
plt.ylabel('$y$', fontsize=14)
plt.title('Phase Portrait', fontsize=18)
plt.legend(fontsize=12)
plt.grid(True)
plt.show()
```

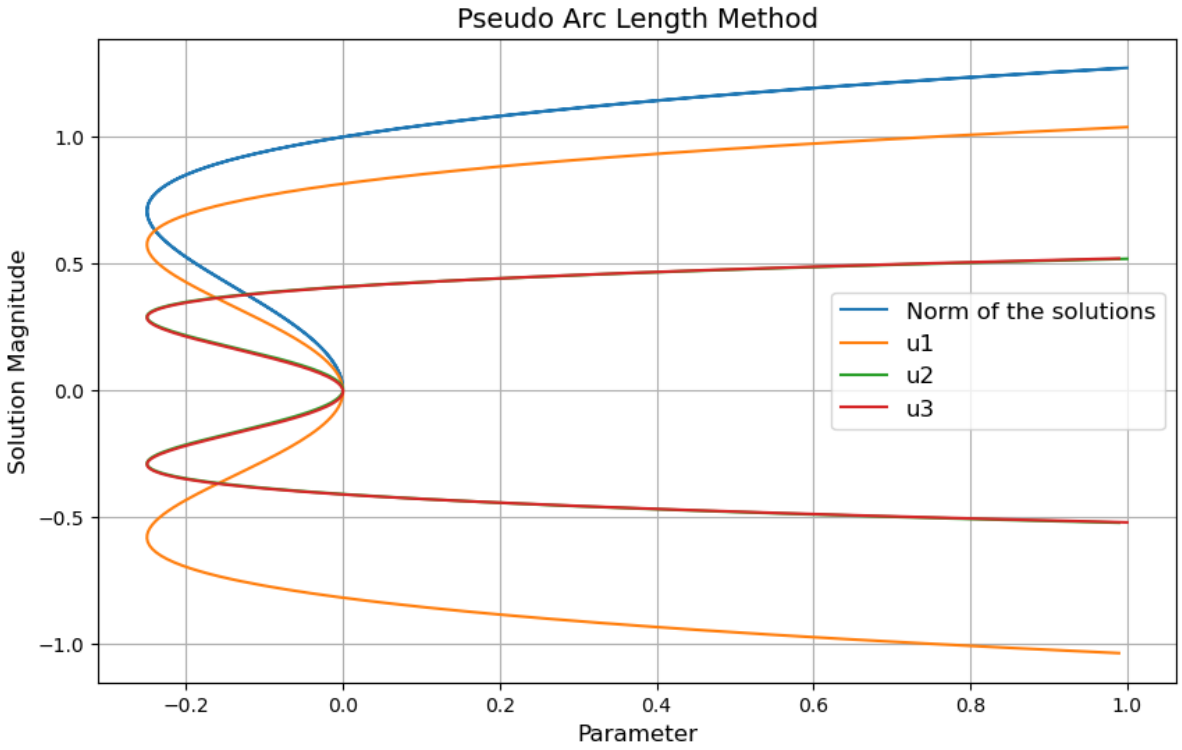
```
The starting point: 1.04, 0.52, -0.52 values and period: 3.63 for the Hopf
Bifurcation 3D orbit
```



When implementing pseudo arc length continuation, for systems of equations which contain squared terms, for example: hopf normal, overflow errors, NaN vlaues and convergence errors may be encountered during shooting. Therefore careful selection of initial conditons is needed. The higher the max steps, the more accurate the result. However, this will come with computational costs, therefore taking longer to run. `pseudo_plotter` works similarly to `natural plotter`.

In [ ]:
```
# (c)
from  numerical_continuation import pseudo_plotter
x0 = sol_lm #starting point from b)
par_array = [1]  # parameter value - must be the same as Max or Min bounds c
par_index = 0

par_pseudo, sol_pseudo = numerical_continuation(hopf_bifurcation_3d,
    'pseudo',
    x0,
    par_array,
    par_index,
    [1, -0.6], #Bounds affect the step size
    [100, 30], #Max steps: [PAL, Natural]
    shoot,
    fsolve,
    phase_condition=phase_condition,
    increase=False)

pseudo_plotter(par_pseudo, sol_pseudo, period=True)
#Takes approx. 2 mins to run, please be patient
```

```
Parameter boundary reached or exceeded.
```

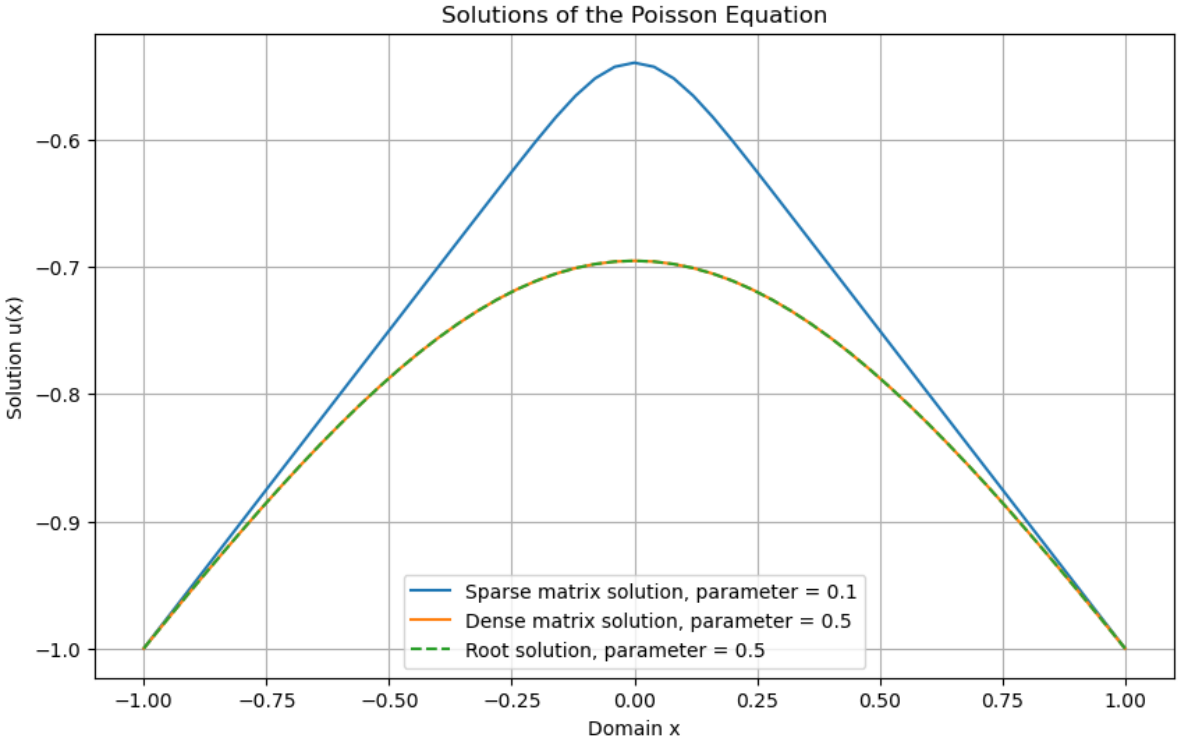


To solve the poisson equation, we import `solve_equation` and `solve_bvp_root` from `sparse_dense_bvp`. Source terms need to be defined, the poisson is defined in `Equations_Functions` as `setup_rhs_poisson`. This can be defined by the user as a normal rhs source term, ensure the signs are correct. The `solve_equation` function solves the PDE using the given discretization and boundary conditions using sparse or dense storage. The solvers use a coefficients dictionary to pass in parameters. `solve_bvp_root` uses `fsolve` to solve a system of ODEs rather than numpy arrays. The equation_type argument allows the solver to solve different types of equations, by modifying the Finite Diff matrix accordingly. For example, if the user wanted to solve a diffusion-convection equation, they would input 'diffusion-convection'.

In [ ]:
```
#Question 3:

from sparse_dense_bvp import solve_equation, solve_bvp_root, q_func
import numpy as np
from Diffusion_OO import BoundaryCondition
import time
import matplotlib.pyplot as plt

from Equations_Functions import setup_rhs_poisson

def setup_rhs_poisson(n_points, coefficients,domain):

    sigma = coefficients.get('sigma')
    rhs = -(1 / (np.sqrt(2 * np.pi * sigma**2))) * np.exp(-domain**2 / (2 *
    return rhs


no_points = 51
a = -1
b = 1
x = np.linspace(a, b, no_points)  # Domain
dx = x[1] - x[0]  # Step size
```

```
boundary_conditions = [
    BoundaryCondition('left', 'dirichlet', -1, coefficients=None),
    BoundaryCondition('right', 'dirichlet', -1, coefficients=None)
]

bc_left = boundary_conditions[0]
bc_right = boundary_conditions[1]

coefficients_possion_dense = {'D': 1.0, 'sigma': 0.5}
coefficients_possion_sparse = {'D': 1.0, 'sigma': 0.1}

x, u_root = solve_bvp_root(no_points, a, b, coefficients_possion_dense['D'],
u_dense= solve_equation('dense', setup_rhs_poisson, x, dx, bc_left, bc_right
u_sparse = solve_equation('sparse', setup_rhs_poisson, x, dx, bc_left, bc_r

print(f'Value of u(0) for dense: {u_dense[25]:.4f}')
print(f'Value of u(0) for sparse: {u_sparse[25]:.4f}')

plt.figure(figsize=(10, 6))
plt.plot(x, u_sparse, label=f'Sparse matrix solution, parameter = {coefficie
plt.plot(x, u_dense, label=f'Dense matrix solution, parameter = {coefficient
plt.plot(x, u_root, label=f'Root solution, parameter = {coefficients_possion
plt.title(f'Solutions of the Poisson Equation')
plt.xlabel('Domain x')
plt.ylabel('Solution u(x)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Value of u(0) for dense: -0.6951
Value of u(0) for sparse: -0.5394
```



In [ ]:
```
#b)

no_points = 501
x = np.linspace(a, b, no_points)  # 501 points in the domain
dx = x[1] - x[0]  # Step size
dx = (b-a)/(no_points-1)
```

```
coefficients = {'D': 1.0, 'sigma': 0.05}

print('Dense Matrix time:')
%timeit u_dense = solve_equation('dense',setup_rhs_poisson,domain=x, h=dx,b
print('Sparse Matrix time:')
%timeit u_sparse = solve_equation('sparse',setup_rhs_poisson,domain=x,  h=dx
```

```
Dense Matrix time:
6.88 ms ± 1.23 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
Sparse Matrix time:
363 µs ± 26.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

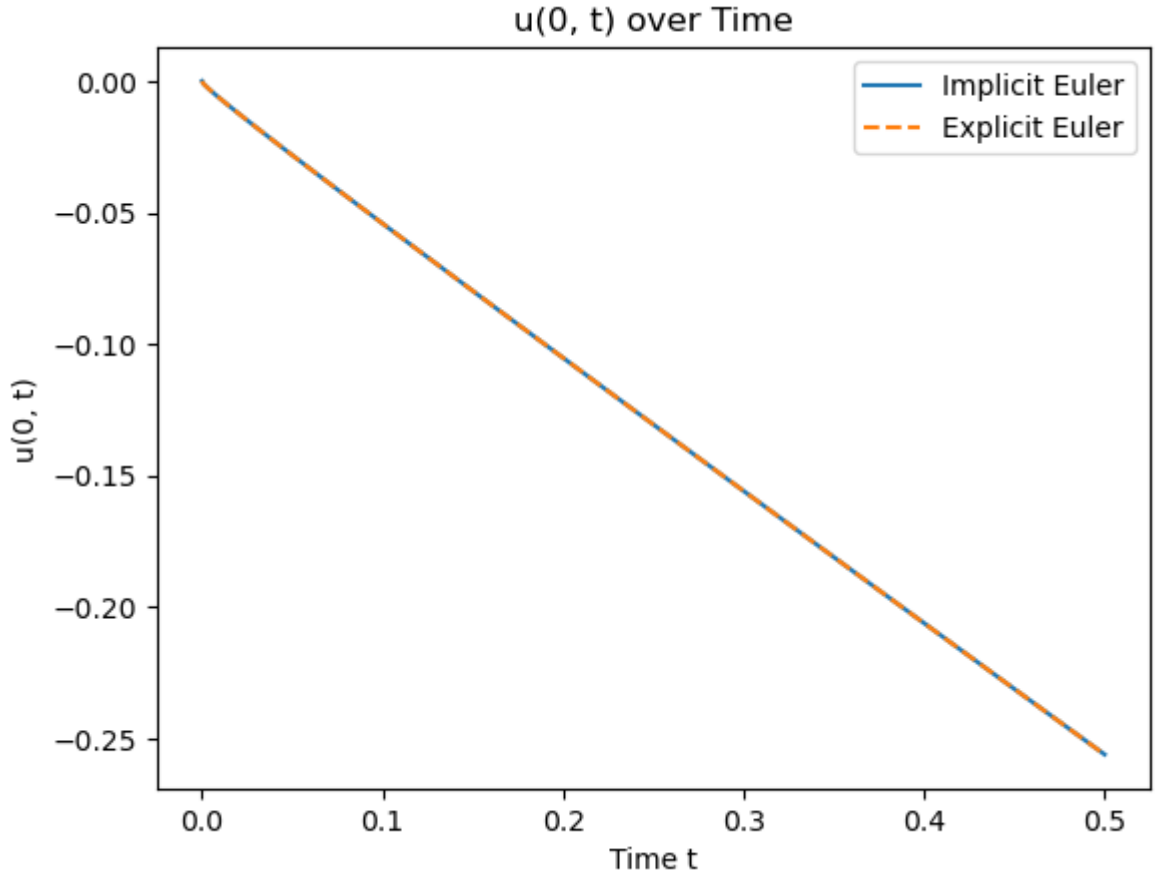
The sparse storage method is significantly faster than the dense storage method. This is typical for sparse storage techniques when using a large N, because sparse solvers (spsolve in this case) are optimised to handle non-zero data more efficiently. This is accomplished by skipping over 0s in the finite difference matrix A. This reduces computational cost, memory and therefore time. Its is approximately 100 times faster. However, the variability in runtime is high for the dense method due to cached memory, which means the sparse method could be faster than 100x.

`Diffusion_OO` uses 2 classes, `BoundaryCondition` and `DiffusionSimulation` to compute solutions to time variant PDEs. `DiffusionSimulation` is a class which takes inputs: source term, a, b, D, IC, BC, N, tspan, method and dt. It initialises these values and then uses the `DiffusionSimulation.solve()` class method to solve depending on the user defined method ['explicit_euler', 'implicit_sparse','implicit_dense','IMEX','implicit_root']. The x descritization is defined inside the class, using the domain boundaries. The source term and initial condition must be callable functions.

In [ ]:
```
# Question 4
from Diffusion_OO import DiffusionSimulation, BoundaryCondition
import numpy as np
import matplotlib.pyplot as plt

boundary_conditions = [
    BoundaryCondition('left', 'neumann', 1),
    BoundaryCondition('right', 'dirichlet', 0)
]

a = 0
b = 2
D = 0.5
N = 101
T = 0.5

def source_term(t, x, U):
    return 0

#Source term defined in the class
initial_condition = lambda x: 0.5*x*(2-x)
dt_max = ((b-a)/(N-1))**2/ (2 * D)

dt = dt_max
print(f"dt MAX ={dt}")
explicit_simulation = DiffusionSimulation(source_term,a, b, D, initial_condi
x_euler, t_eval_euler, U_euler = explicit_simulation.solve()
```

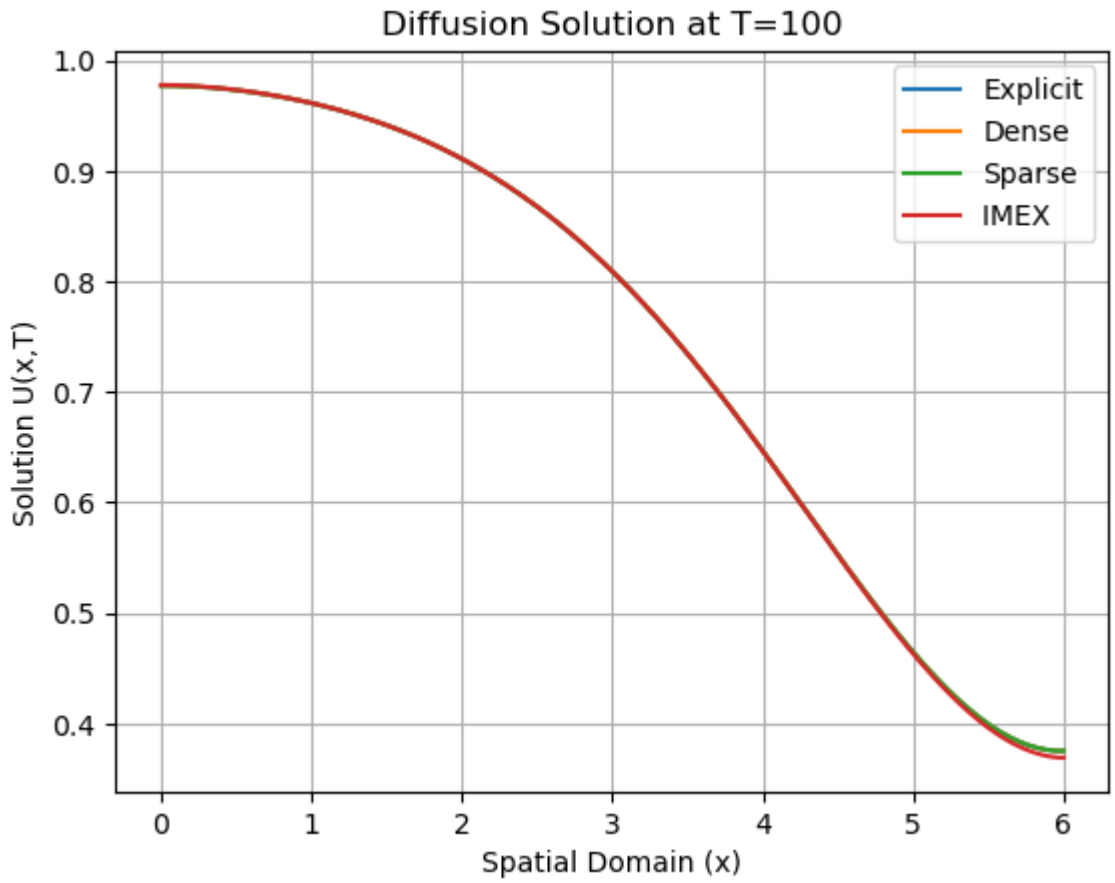
```
implicit_simulation = DiffusionSimulation(source_term,a, b, D, initial_condi
x_imp, t_eval_imp, U_imp = implicit_simulation.solve()

u_exp_x0 = U_euler[:, 0] #extract solution at x = 0 for all times
print(f"u(0, T) at T = {t_eval_euler[-1]} is: {u_exp_x0[-1]}")

u_imp_x0 = U_imp[:, 0] #extract solution at x = 0 for all times
print(f"u(0, T) at T = {t_eval_imp[-1]} is: {u_imp_x0[-1]}")

plt.plot(t_eval_imp, u_imp_x0, label='Implicit Euler')
plt.plot(t_eval_euler, u_exp_x0, label='Explicit Euler',linestyle ='--') #pl
plt.title('u(0, t) over Time')
plt.xlabel('Time t')
plt.ylabel('u(0, t)')
plt.legend()
plt.show()
```

```
dt MAX =0.0004
/Users/edatkinson/anaconda3/lib/python3.11/site-packages/scipy/sparse/_inde
x.py:143: SparseEfficiencyWarning: Changing the sparsity structure of a csr
_matrix is expensive. lil_matrix is more efficient.
  self._set_arrayXarray(i, j, x)
/Users/edatkinson/EMAt30008/SciComp/Diffusion_OO.py:174: RuntimeWarning: Us
ing a sparse solver for a small system may not be optimal.
  warnings.warn("Using a sparse solver for a small system may not be optima
l.", RuntimeWarning)
u(0, T) at T = 0.5 is: -0.2560309173893463
u(0, T) at T = 0.5 is: -0.2560205512463996
```



In [ ]:
```
#Question 5:
from Diffusion_OO import DiffusionSimulation, BoundaryCondition
import numpy as np
import matplotlib.pyplot as plt
import time
```

```
boundary_conditions = [
    BoundaryCondition('left', 'neumann', 0, coefficients=None),
    BoundaryCondition('right', 'neumann', 0, coefficients=None)
]

a = 0
b = 6
D = 0.01
N = 200
T = 100

def source_term(t, x, U):
    return ((1-U)**2)*np.exp(-x)

initial_condition = lambda x: x*0

dt_max = ((b-a)/N)**2/ (2 * D)
dt = dt_max

#Explicit Euler
exp_simulation = DiffusionSimulation(source_term,a, b, D, initial_condition,
x_exp, t_exp, U_exp = exp_simulation.solve()
#Implicit Euler Dense
imp_dense_simulation = DiffusionSimulation(source_term,a, b, D, initial_cond
x_imp_dense, t_imp_dense, U_imp_dense = imp_dense_simulation.solve()
#Implicit Euler Sparse
imp_sparse_simulation = DiffusionSimulation(source_term,a, b, D, initial_co
x_imp_sparse, t_imp_sparse, U_imp_sparse = imp_sparse_simulation.solve()
#IMEX
imp_root_simulation = DiffusionSimulation(source_term,a, b, D, initial_condi
x_imp_root, t_imp_root, U_imp_root = imp_root_simulation.solve()

plt.plot(x_exp, U_exp[-1], label='Explicit')
plt.plot(x_imp_dense, U_imp_dense[-1], label='Dense')
plt.plot(x_imp_sparse, U_imp_sparse[-1], label='Sparse')
plt.plot(x_imp_root, U_imp_root[-1], label='IMEX ')


plt.xlabel('Spatial Domain (x)')
plt.ylabel('Solution U(x,T)')
plt.title('Diffusion Solution at T=100')
plt.legend()
plt.grid(True)
plt.show()
```



In [ ]:
```
#b)
print('Explicit time: ')
%timeit x_exp, t_exp, U_exp = exp_simulation.solve()
print('Dense Implicit time: ')
%timeit x_imp_dense, t_imp_dense, U_imp_dense = imp_dense_simulation.solve(
print('Sparse Implicit time: ')
%timeit x_imp_sparse, t_imp_sparse, U_imp_sparse = imp_sparse_simulation.so
print('IMEX Implicit time: ')
%timeit x_imp_root, t_imp_root, U_imp_root = imp_root_simulation.solve()
#IMEX takes the longest time to run.
```

```
Explicit time:
72.9 ms ± 3.95 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
Dense Implicit time:
1.32 s ± 227 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Sparse Implicit time:
187 ms ± 7.34 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
IMEX Implicit time:
22.1 s ± 269 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The method I'd choose purely based off of time is the Explicit Euler method. It has the lowest average run time. However, varying the parameter D will influence the max step size, dt_max. The larger D becomes, the smaller the t_max, therefore if D is being increased the method will slow down. As a result of this explicit euler method may not be optimal. Therefore I propose using implicit euler with sparse storage. This is the next fastest method, averaging approx 181ms per run. Furthermore, if the size of N is increased, the sparse solver will still be able to run quickly due to how spsolve optimises matrix multiplications. Implicit Euler is unconditionally stable, meaning any step size dt can be used. Therefore, I would opt to use the Implicit Euler sparse solver.

In [ ]:
```
# c)
print(f"\nValue of u(L, T) for Implicit Euler Sparse: {U_imp_sparse[-1][-1]
```

```
Value of u(L, T) for Implicit Euler Sparse: 0.3752
```

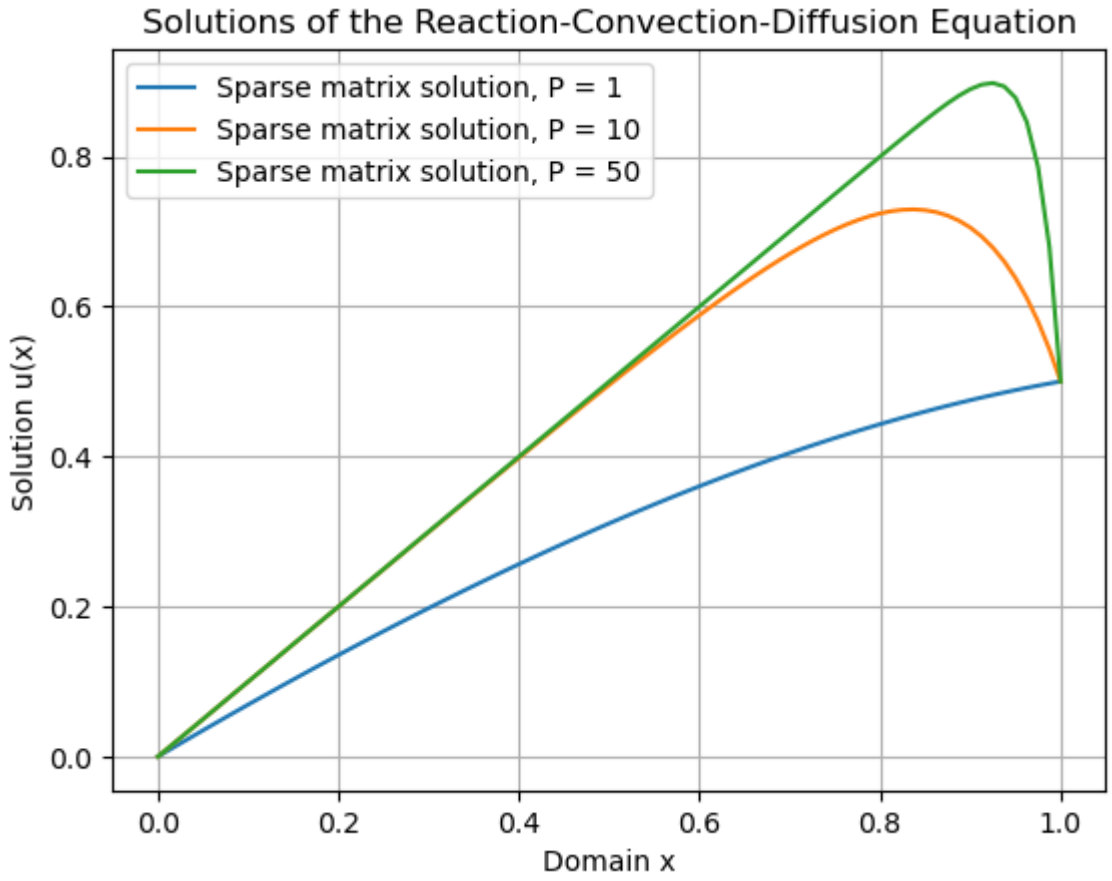
part c I've chosen this method because it is unconditionally stable. Implicit Euler methods can be used to solve non-linear PDEs, which is the case in this problem. It produces the same result as the other methods, meaning they are all accurate. So I decided to use this method as it is the quickest and doesn't rely on a small time step.

Question 6: Uses `sparse_dense_bvp` to solve this convection-diffusion equation. Setting up the problem is the same as in question 3. Except the equation type is 'convection-diffusion'. The rhs function is a vector of -P. The finite difference matrix is modified accordingly to handle the 1st order derivative, so this is not required in the rhs function.

In [ ]:
```
#Question 6
#a)
import numpy as np
import matplotlib.pyplot as plt
from sparse_dense_bvp import solve_equation
from Diffusion_OO import BoundaryCondition
from Equations_Functions import setup_rhs_reaction #RHS function


def setup_rhs_reaction(n_points, coefficients, domain):
    P = coefficients.get('P')
    rhs = -np.ones(n_points) * P
    return rhs

equation_type_Q6 = 'convection-diffusion' #need to set up a rhs function fo

no_points = 81
a = 0
b = 1
x = np.linspace(a, b, no_points)  # 501 points in the domain
dx = x[1] - x[0]  # Step size
dx = (b-a)/(no_points-1)


boundary_conditions = [
BoundaryCondition('left', 'dirichlet', 0, coefficients=None),
BoundaryCondition('right', 'dirichlet', 0.5, coefficients=None)
]


bc_left = boundary_conditions[0]
bc_right = boundary_conditions[1]

#P = 1
coefficients_P1 = {'P': 1}
u_sparse_1 = solve_equation('sparse',setup_rhs_reaction,domain=x, h=dx, bc_
print(f"Max solution: {max(u_sparse_1):5f}")


#P = 10
coefficients_P2 = {'P': 10}
u_sparse_2 = solve_equation('sparse',setup_rhs_reaction,domain=x, h=dx, bc_
```

```
print(f"Max solution: {max(u_sparse_2):5f}")


#P = 50
coefficients_P3 = {'P': 50}
u_sparse_3 = solve_equation('sparse',setup_rhs_reaction,domain=x, h=dx, bc_
print(f"Max solution: {max(u_sparse_3):5f}")

plt.plot(x, u_sparse_1, label=f'Sparse matrix solution, P = {coefficients_P1
plt.plot(x, u_sparse_2, label=f'Sparse matrix solution, P = {coefficients_P2
plt.plot(x, u_sparse_3, label=f'Sparse matrix solution, P = {coefficients_P3
plt.title(f'Solutions of the Reaction-Convection-Diffusion Equation')
plt.xlabel('Domain x')
plt.ylabel('Solution u(x)')
plt.legend()
plt.grid(True)
plt.show()
```

```
Max solution: 0.500000
Max solution: 0.729391
Max solution: 0.897845
```

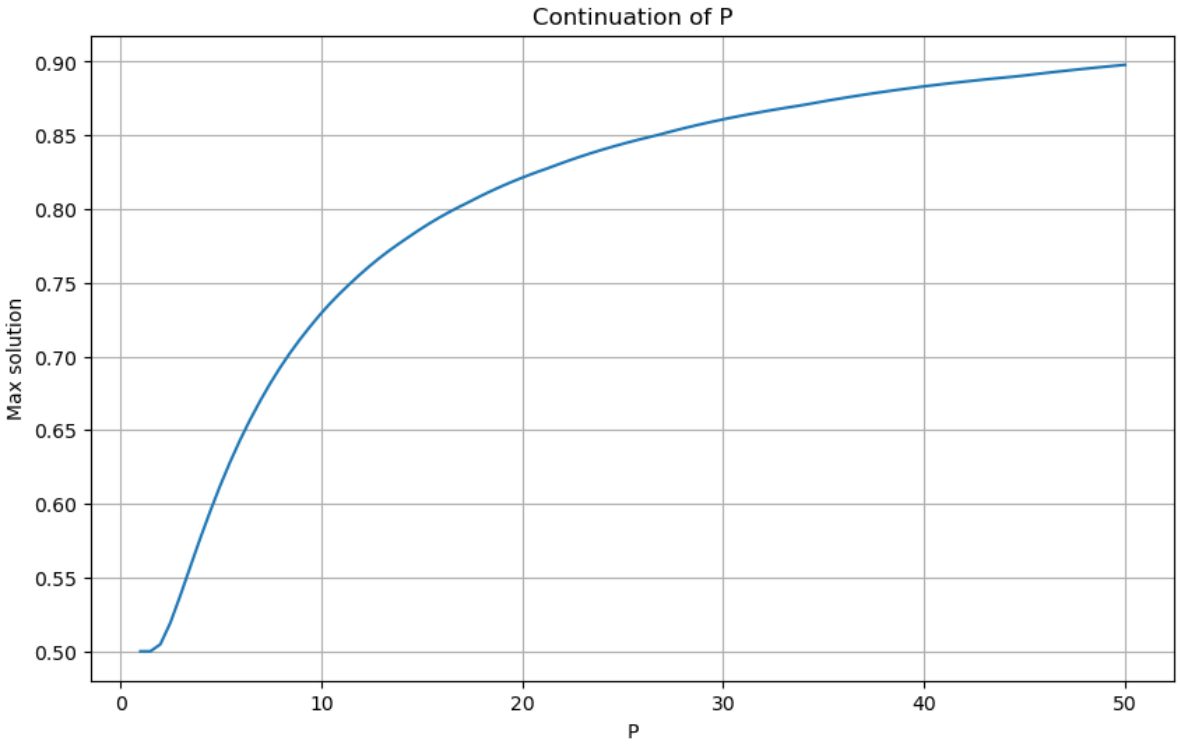


Uses a simple natural parameter continuation, to solve the system over a range of parameters 1 < P < 50.

In [ ]:

```
#b)

def pde_continuation(rhs,par_bounds, max_steps):
    pars = np.linspace(par_bounds[0], par_bounds[1], max_steps)
    sols = []
    for i in range(max_steps):
        U_j = solve_equation('sparse',rhs,domain=x, h=dx, bc_left=bc_left, b
        sols.append(U_j)
    return pars, sols
```

```
par_bounds = [1, 50]
max_steps = 100
pars, sols = pde_continuation(setup_rhs_reaction,par_bounds, max_steps)
plt.figure(figsize=(10, 6))
plt.plot(pars, [max(sol) for sol in sols])
plt.xlabel('P')
plt.ylabel('Max solution')
plt.title('Continuation of P')
plt.grid(True)
plt.show()
```



Key Software Decisions

Blanket Decisions:

Decisions that are consistent across all my code include consideration of modularity, error handling and efficiency. I've tried to maintain a good degree of modularity for each numerical method I implement. This approach significantly aids in debugging and troubleshooting. Enhancing modularity allowed me to pinpoint problem sources more efficiently, rather than combing through large blocks of code. Additionally, modularity enhances the readability of the code, as it allows each function to focus on a specific task. For error handling, I incorporate error checks, `ValueError`, `TypeError` handling, `try-except` blocks, and `warnings`. This guides users through potential issues with their inputs and adds a level of gracefulness to how errors are managed. I employ the `warnings` and `logging` libraries for this purpose. Logging is particularly useful as it captures and reports the state of the system at the point an error occurs, diagnosing issues and formulating solutions. Furthermore, each function is documented with a comprehensive API in the form of a docstring. This documentation specifies the

expected inputs and their types, providing guidance for users on how to interact with the functionality effectively.

Modules:

Solving ODEs:

- Euler's method and the 4th order Runge-Kutta (RK4) method are step functions and are employed within the `solve_to` function. This function uses an initial condition to solve an ordinary differential equation (ODE) over a specified time interval. The core ODE solver used in numerical shooting and continuation, `solve_ode` employs a for-loop to apply these step functions iteratively across the time domain. For enhanced efficiency, solutions are stored directly in a 2 dimensional pre-allocated numpy array, which is computationally more efficient than dynamically appending to arrays or lists. The `solve_ode` function is capable of handling ODEs of any dimension and employs `*args` to pass various parameters into the system. The use of `*args` provides the flexibility to accommodate additional parameters required by different systems of equations. Comparative accuracy tests between the Euler method and the RK4 method across a broad range of step sizes revealed that RK4 significantly outperforms the Euler method, achieving accuracies up to 1e-11, compared to 1e−3 for the Euler method, with a step size of 1e−3. Consequently, the RK4 method is the preferred choice for use in shooting and other numerical methods. Additionally, the solver includes a simple adaptive step size feature to manage potential overflow issues, ensuring robust and reliable computation under varying conditions.

Bvp and Shooting:

- The numerical shooting software solves boundary value ODE problems, it uses `solve_ode` and `scipy.optimize.fsolve` for root finding. The `shoot` function implements numerical shooting for periodic solutions to an ODE by converting a boundary value problem into an initial value problem. This is passed into the root finder, located in the `limit_cycle_finder` function, to isolate a limit cycle based on an initial condition. Works by adjusting the initial condition and T, until the end of one period matches the start. To trace the orbit of the limit cycle, the `orbit` function is used to solve the ODE at the Isolated Limit Cycle state found by `limit_cycle_finder`. `shoot` uses `np.isnan` to check for NaN values and ensure `solve_ode` doesn't return unexpected solutions.

Numerical Continuation:

- One function, `numerical_continuation`, is called to carry out numerical continuation. There are two methods, Natural and Pseudo Arc-Length (PAL). Natural Continuation is straightforward, it iterates over a parameter range using a for loop. Depending on if a phase condition is provided it can track limit cycles or equilibria - there is no modularity here as it is a simple numerical method. For efficiency, solutions are allocated to a pre-allocated numpy array to store solutions. Pseudo

Arc-Length works a little differently. Natural Parameter continuation was used in `find_initial_sols` to find two initial solutions, using a step size calculated by taking the difference between the Max and Min Parameters divided by the Max steps. I employed a while loop which used the root finder to find the corrected solutions. A while loop was used because PAL calculates the value of the next parameter, so in order to continue over the whole parameter range, the loop would need to iterate until the value of the corrected parameter is out of the parameter range. Using a for loop wouldn't cover the whole range as we don't know how many steps the continuation will need- this depends on the ODE system being continued. PAL continuation has modularity, this is because there are multiple processes which can be divided into multiple simple functions which carry out specific purposes. In the `predict` function, I used vectors in the form [u1,...,u_n,T,P] and performed elementwise operations to calculate and extract U_pred, P_pred and secants for the PAL constraint - the use of extraction is more efficient than splitting up the initial vector and defining new variables.

Time Invariant BVP Solver:

- Located in `sparse_dende_bvp.py`. Within `solve_equation` Each function handles specific aspects of differential equation solutions - such as matrix setup, boundary condition application, and solution methods - enhancing readability and maintainability. The numerical method supports both sparse and dense matrix operations, offering users the choice to select the method best suited to the problem's characteristics and computational resources. Flexibility is further enhanced by using Python dictionaries for passing coefficients and Callable for defining right-hand side functions, allowing adaptation to different types of differential equations without altering code. The setup of a `BoundaryCondition` class in `Diffusion_OO` means that boundary objects can be created easily and integrated into this solver. Dynamic application of boundary conditions showcases adaptability to diverse physical scenarios, backed by input validation to ensure computational reliability and correctness. Python features, such as `typing` annotations, are employed to improve code clarity and facilitate error checking during development. Collectively, these design choices reflect a commitment to creating a robust, adaptable, and user-friendly numerical toolkit capable of handling everything from simple diffusion problems to complex convection-diffusion equations.

Time Variant BVP solver:

- Located in `Diffusion_OO`. In developing the `DiffusionSimulation` class, key design decisions were made to enhance the software's robustness, scalability, and ease of use. A principle decision was the adoption of an object-oriented design, which facilitated encapsulation of all diffusion-related properties and methods into a single cohesive unit. This design promotes modularity—allowing each component to be developed and tested independently which improves the code's maintainability. By encapsulating all the different methods for solving PDEs, for example

`implicit_euler_step`, `explicit_euler_step` and others, the class structure provides a clear and intuitive interface for users while maintaining a high degree of flexibility. The class design allows for easy adjustments or extensions, such as adding new types of boundary conditions or integration methods, without altering the existing codebase. Furthermore, the use of `assert` statements for input validation and comprehensive error handling within each method ensures that the system behaves predictably under various input conditions and provides useful feedback, thereby preventing runtime errors and enhancing user experience.

Testing

- I've employed a range of code tests, which range from comparing the numerical solutions to analyical solutions, type checks, value checks, input validations, and others specific to the numerical methods. Although the code isn't fully tested, the most important areas of each method is tested to a good standard. I use the `unittest` library to run my tests, as this provided a class based approach to testing, which I found to be easy to define and organise my tests.

Self Reflection Report

Reflecting on this unit, I have garnered substantial insights into both mathematical algorithms and software engineering, identifying crucial learning moments and future directions for growth. Here's a comprehensive self-reflection based on my experiences:

What did I learn about the mathematical algorithms?

- My engagement with ODEs and PDEs deepened significantly, bridging theoretical knowledge with practical numerical methods. This integration enhanced my understanding of mathematical concepts and their applications.

- Implementing numerical shooting to solve BVPs and iterative refinement processes were highlights. Going beyond and understanding how shooting can be applied to real world scenarios helped to solidify my understanding of the numerical process involved with shooting. Additionally, adopting matrix methods for PDEs, though initially challenging, provided valuable insights into complex mathematical operations and their real-world applications. I liked how converting the method of lines from a system of equations to a matrix representation to be solved using matrix operations is more efficient due to how computers can handle matrix multiplication much faster, allowing for clearer more efficient code.

What did I learn about software engineering?

- Initially, my coding practices were non-existent, leading to complex debugging processes. Over time, adopting principles like DRY (Don't Repeat Yourself), modularization, and abstraction improved my code's organization and maintainability, facilitating easier updates and debugging. These were especially

helpful when building on top of old code - something which was very common in this course. It helped with identifying flaws which led to better design decisions.

- Using APIs and incorporating docstrings enhanced manageability and comprehension of my code's functionality, crucial for both immediate understanding and future modifications.

- Learning to use `*args` for dynamic parameter passing in functions was a breakthrough, offering enhanced flexibility in function design and problem-solving.

- Comprehensively testing my code seemed like alot of effort at first. However, after adopting the unittests library, it became easy to define and organise my code tests. These became crucial for developing my code, as it would provide me with indications about why my code is failing and what can be done to mitigate those problems.

- Profiling to locate bottlenecks was helpful to increase the efficiency of my software. One notable improvement I made was during the sparse matrix setup process. I initally used `scipy.sparse.csr_matrix.tocsr` to convert a numpy array into a `csr` formatted matrix. This was less efficient than initially setting up the matrix using `scipy.sparse.diags`.

- Git is second nature now. I've made good use of the git and tried to make regular small commits. Sometimes I'd forget to commit some of my code, meaning I would end up committing large chunks of code. And using `git add –all` commands, which means I lose out on detailing changes I have made to each file. I could have been more consistent with descriptions in my commits, as sometimes I'd commit a full explanation and other times it would be a rough description. I am happy with the skills I've learnt from using git.

What would I have done differently if I started the unit over again?

- Strategic Approach: If I were to start over, I would integrate a class-based approach to manage grids, matrices and solvers, aiming to streamline processes and reduce redundancy, particularly in applying boundary conditions.

- Commitment to Code Management: Improving my commit habits to include more frequent updates with detailed messages would enhance change tracking and project management. Furthermore, using branches to single out features of my code and develop them fully before merging them with the rest of the code. This would be especially beneficial for group work - which is the case for most software projects.

What will I do differently in the future as a result of this unit?

- I plan to make smaller, more frequent commits with comprehensive descriptions to better document code evolution, enhancing project management and minimizing oversight.

- The challenges and lessons from this unit inspire me to explore advanced programming techniques and sophisticated numerical methods, potentially helping me with data-driven physical modelling next year.

This reflection underscores the technical skills developed and illustrates my proactive approach to continuous improvement in integrating mathematical theory with practical software applications. The unit has significantly contributed to my knowledge and methodological approaches in solving complex problems. I spent a lot of time developing this software. I grew to enjoy this unit, no matter how confusing my results were. My coding has improved tremendously and I am forever happy that I took this unit as I have gained some valuable skills.

In [ ]:
```
#Prints word count of markdown cells:
import io
import nbformat

with io.open('/Users/edatkinson/EMAt30008/SciComp/Report.ipynb', 'r', encod
    nb = nbformat.read(f, as_version=4)

indices_of_interest = {27,28}

for index in indices_of_interest:
    cell = nb['cells'][index]
    if cell['cell_type'] == "markdown":
        word_count = len(cell['source'].replace('#', '').lstrip().split(' '
        print(f"Word count in cell {index+1}: {word_count}")
```

```
Word count in cell 28: 1221
Word count in cell 29: 749
```