Solving nonlinear ODE and PDE problems

Hans Petter Langtangen[1,2]

Svein Linge[3,1]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo
[3]Department of Process, Energy and Environmental Technology, University College of Southeast Norway

Aug 15, 2017

Note: Preliminary version (expect typos).

Contents

List of Exercises and Problems

1 Introduction of basic concepts

1.1 Linear versus nonlinear equations

Algebraic equations. A linear, scalar, algebraic equation in x has the form

$$ax + b = 0,$$

for arbitrary real constants a and b. The unknown is a number x. All other algebraic equations, e.g., $x^2 + ax + b = 0$, are nonlinear. The typical feature in a nonlinear algebraic equation is that the unknown appears in products with itself, like x^2 or $e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{3!}x^3 + \cdots$.

We know how to solve a linear algebraic equation, $x = -b/a$, but there are no general methods for finding the exact solutions of nonlinear algebraic equations, except for very special cases (quadratic equations constitute a primary example). A nonlinear algebraic equation may have no solution, one solution, or many solutions. The tools for solving nonlinear algebraic equations are *iterative methods*, where we construct a series of linear equations, which we know how to solve, and hope that the solutions of the linear equations converge to a solution of the nonlinear equation we want to solve. Typical methods for nonlinear algebraic equation equations are Newton's method, the Bisection method, and the Secant method.

Differential equations. The unknown in a differential equation is a function and not a number. In a linear differential equation, all terms involving the unknown function are linear in the unknown function or its derivatives. Linear here means that the unknown function, or a derivative of it, is multiplied by a number or a known function. All other differential equations are non-linear.

The easiest way to see if an equation is nonlinear, is to spot nonlinear terms where the unknown function or its derivatives are multiplied by each other. For example, in

$$u'(t) = -a(t)u(t) + b(t),$$

the terms involving the unknown function u are linear: u' contains the derivative of the unknown function multiplied by unity, and au contains the unknown function multiplied by a known function. However,

$$u'(t) = u(t)(1 - u(t)),$$

is nonlinear because of the term $-u^2$ where the unknown function is multiplied by itself. Also

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0,$$

is nonlinear because of the term uu_x where the unknown function appears in a product with its derivative. (Note here that we use different notations for

derivatives: u' or du/dt for a function $u(t)$ of one variable, $\frac{\partial u}{\partial t}$ or u_t for a function of more than one variable.)

Another example of a nonlinear equation is

$$u'' + \sin(u) = 0,$$

because $\sin(u)$ contains products of u, which becomes clear if we expand the function in a Taylor series:

$$\sin(u) = u - \frac{1}{3}u^3 + \ldots$$

Mathematical proof of linearity.

To really prove mathematically that some differential equation in an unknown u is linear, show for each term $T(u)$ that with $u = au_1 + bu_2$ for constants a and b,

$$T(au_1 + bu_2) = aT(u_1) + bT(u_2)\,.$$

For example, the term $T(u) = (\sin^2 t)u'(t)$ is linear because

$$\begin{aligned} T(au_1 + bu_2) &= (\sin^2 t)(au_1(t) + bu_2(t)) \\ &= a(\sin^2 t)u_1(t) + b(\sin^2 t)u_2(t) \\ &= aT(u_1) + bT(u_2)\,. \end{aligned}$$

However, $T(u) = \sin u$ is nonlinear because

$$T(au_1 + bu_2) = \sin(au_1 + bu_2) \neq a\sin u_1 + b\sin u_2\,.$$

1.2 A simple model problem

A series of forthcoming examples will explain how to tackle nonlinear differential equations with various techniques. We start with the (scaled) logistic equation as model problem:

$$u'(t) = u(t)(1 - u(t))\,. \tag{1}$$

This is a nonlinear ordinary differential equation (ODE) which will be solved by different strategies in the following. Depending on the chosen time discretization of (1), the mathematical problem to be solved at every time level will either be a linear algebraic equation or a nonlinear algebraic equation. In the former case, the time discretization method transforms the nonlinear ODE into linear subproblems at each time level, and the solution is straightforward to find since

linear algebraic equations are easy to solve. However, when the time discretization leads to nonlinear algebraic equations, we cannot (except in very rare cases) solve these without turning to approximate, iterative solution methods.

The next subsections introduce various methods for solving nonlinear differential equations, using (1) as model. We shall go through the following set of cases:

- explicit time discretization methods (with no need to solve nonlinear algebraic equations)
- implicit Backward Euler time discretization, leading to nonlinear algebraic equations solved by
  - an exact analytical technique
  - Picard iteration based on manual linearization
  - a single Picard step
  - Newton's method
- implicit Crank-Nicolson time discretization and linearization via a geometric mean formula

Thereafter, we compare the performance of the various approaches. Despite the simplicity of (1), the conclusions reveal typical features of the various methods in much more complicated nonlinear PDE problems.

1.3 Linearization by explicit time discretization

Time discretization methods are divided into explicit and implicit methods. Explicit methods lead to a closed-form formula for finding new values of the unknowns, while implicit methods give a linear or nonlinear system of equations that couples (all) the unknowns at a new time level. Here we shall demonstrate that explicit methods constitute an efficient way to deal with nonlinear differential equations.

The Forward Euler method is an explicit method. When applied to (1), sampled at $t = t_n$, it results in

$$\frac{u^{n+1} - u^n}{\Delta t} = u^n(1 - u^n),$$

which is a *linear* algebraic equation for the unknown value u^{n+1} that we can easily solve:

$$u^{n+1} = u^n + \Delta t\, u^n(1 - u^n)\,.$$

In this case, the nonlinearity in the original equation poses no difficulty in the discrete algebraic equation. Any other explicit scheme in time will also give only linear algebraic equations to solve. For example, a typical 2nd-order Runge-Kutta method for (1) leads to the following formulas:

$$u^* = u^n + \Delta t u^n(1-u^n),$$
$$u^{n+1} = u^n + \Delta t \frac{1}{2}\left(u^n(1-u^n) + u^*(1-u^*)\right)\,.$$

The first step is linear in the unknown u^*. Then u^* is known in the next step, which is linear in the unknown u^{n+1} .

1.4 Exact solution of nonlinear algebraic equations

Switching to a Backward Euler scheme for (1),

$$\frac{u^n - u^{n-1}}{\Delta t} = u^n(1-u^n), \tag{2}$$

results in a nonlinear algebraic equation for the unknown value u^n. The equation is of quadratic type:

$$\Delta t(u^n)^2 + (1-\Delta t)u^n - u^{n-1} = 0,$$

and may be solved exactly by the well-known formula for such equations. Before we do so, however, we will introduce a shorter, and often cleaner, notation for nonlinear algebraic equations at a given time level. The notation is inspired by the natural notation (i.e., variable names) used in a program, especially in more advanced partial differential equation problems. The unknown in the algebraic equation is denoted by u, while $u^{(1)}$ is the value of the unknown at the previous time level (in general, $u^{(\ell)}$ is the value of the unknown ℓ levels back in time). The notation will be frequently used in later sections. What is meant by u should be evident from the context: u may either be 1) the exact solution of the ODE/PDE problem, 2) the numerical approximation to the exact solution, or 3) the unknown solution at a certain time level.

The quadratic equation for the unknown u^n in (2) can, with the new notation, be written

$$F(u) = \Delta t u^2 + (1-\Delta t)u - u^{(1)} = 0\,. \tag{3}$$

The solution is readily found to be

$$u = \frac{1}{2\Delta t}\left(-1 + \Delta t \pm \sqrt{(1-\Delta t)^2 - 4\Delta t u^{(1)}}\right)\,. \tag{4}$$

Now we encounter a fundamental challenge with nonlinear algebraic equations: the equation may have more than one solution. How do we pick the right solution? This is in general a hard problem. In the present simple case, however, we can analyze the roots mathematically and provide an answer. The idea is to expand the roots in a series in Δt and truncate after the linear term since the Backward Euler scheme will introduce an error proportional to Δt anyway. Using `sympy`, we find the following Taylor series expansions of the roots:

```
>>> import sympy as sym
>>> dt, u_1, u = sym.symbols('dt u_1 u')
>>> r1, r2 = sym.solve(dt*u**2 + (1-dt)*u - u_1, u)  # find roots
>>> r1
(dt - sqrt(dt**2 + 4*dt*u_1 - 2*dt + 1) - 1)/(2*dt)
>>> r2
(dt + sqrt(dt**2 + 4*dt*u_1 - 2*dt + 1) - 1)/(2*dt)
>>> print r1.series(dt, 0, 2)    # 2 terms in dt, around dt=0
-1/dt + 1 - u_1 + dt*(u_1**2 - u_1) + O(dt**2)
>>> print r2.series(dt, 0, 2)
u_1 + dt*(-u_1**2 + u_1) + O(dt**2)
```

We see that the `r1` root, corresponding to a minus sign in front of the square root in (4), behaves as $1/\Delta t$ and will therefore blow up as $\Delta t \to 0$! Since we know that u takes on finite values, actually it is less than or equal to 1, only the `r2` root is of relevance in this case: as $\Delta t \to 0$, $u \to u^{(1)}$, which is the expected result.

For those who are not well experienced with approximating mathematical formulas by series expansion, an alternative method of investigation is simply to compute the limits of the two roots as $\Delta t \to 0$ and see if a limit appears unreasonable:

```
>>> print r1.limit(dt, 0)
-oo
>>> print r2.limit(dt, 0)
u_1
```

1.5 Linearization

When the time integration of an ODE results in a nonlinear algebraic equation, we must normally find its solution by defining a sequence of linear equations and hope that the solutions of these linear equations converge to the desired solution of the nonlinear algebraic equation. Usually, this means solving the linear equation repeatedly in an iterative fashion. Alternatively, the nonlinear equation can sometimes be approximated by one linear equation, and consequently there is no need for iteration.

Constructing a linear equation from a nonlinear one requires *linearization* of each nonlinear term. This can be done manually as in Picard iteration, or fully algorithmically as in Newton's method. Examples will best illustrate how to linearize nonlinear problems.

1.6 Picard iteration

Let us write (3) in a more compact form

$$F(u) = au^2 + bu + c = 0,$$

with $a = \Delta t$, $b = 1 - \Delta t$, and $c = -u^{(1)}$. Let u^- be an available approximation of the unknown u. Then we can linearize the term u^2 simply by writing $u^- u$. The resulting equation, $\hat{F}(u) = 0$, is now linear and hence easy to solve:

$$F(u) \approx \hat{F}(u) = au^- u + bu + c = 0\,.$$

Since the equation $\hat{F} = 0$ is only approximate, the solution u does not equal the exact solution u_e of the exact equation $F(u_\mathrm{e}) = 0$, but we can hope that u is closer to u_e than u^- is, and hence it makes sense to repeat the procedure, i.e., set $u^- = u$ and solve $\hat{F}(u) = 0$ again. There is no guarantee that u is closer to u_e than u^-, but this approach has proven to be effective in a wide range of applications.

The idea of turning a nonlinear equation into a linear one by using an approximation u^- of u in nonlinear terms is a widely used approach that goes under many names: *fixed-point iteration*, the method of *successive substitutions*, *nonlinear Richardson iteration*, and *Picard iteration*. We will stick to the latter name.

Picard iteration for solving the nonlinear equation arising from the Backward Euler discretization of the logistic equation can be written as

$$u = -\frac{c}{au^- + b}, \quad u^- \ \leftarrow \ u\,.$$

The $\leftarrow$ symbols means assignment (we set u^- equal to the value of u). The iteration is started with the value of the unknown at the previous time level: $u^- = u^{(1)}$.

Some prefer an explicit iteration counter as superscript in the mathematical notation. Let u^k be the computed approximation to the solution in iteration k. In iteration $k+1$ we want to solve

$$au^k u^{k+1} + bu^{k+1} + c = 0 \quad \Rightarrow \quad u^{k+1} = -\frac{c}{au^k + b}, \quad k = 0, 1, \ldots$$

Since we need to perform the iteration at every time level, the time level counter is often also included:

$$au^{n,k} u^{n,k+1} + bu^{n,k+1} - u^{n-1} = 0 \quad \Rightarrow \quad u^{n,k+1} = \frac{u^n}{au^{n,k} + b}, \quad k = 0, 1, \ldots,$$

with the start value $u^{n,0} = u^{n-1}$ and the final converged value $u^n = u^{n,k}$ for sufficiently large k.

However, we will normally apply a mathematical notation in our final formulas that is as close as possible to what we aim to write in a computer code and then it becomes natural to use u and u^- instead of u^{k+1} and u^k or $u^{n,k+1}$ and $u^{n,k}$.

Stopping criteria. The iteration method can typically be terminated when the change in the solution is smaller than a tolerance ϵ_u:

$$|u - u^-| \leq \epsilon_u,$$

or when the residual in the equation is sufficiently small ($< \epsilon_r$),

$$|F(u)| = |au^2 + bu + c| < \epsilon_r \,.$$

A single Picard iteration. Instead of iterating until a stopping criterion is fulfilled, one may iterate a specific number of times. Just one Picard iteration is popular as this corresponds to the intuitive idea of approximating a nonlinear term like $(u^n)^2$ by $u^{n-1}u^n$. This follows from the linearization u^-u^n and the initial choice of $u^- = u^{n-1}$ at time level t_n. In other words, a single Picard iteration corresponds to using the solution at the previous time level to linearize nonlinear terms. The resulting discretization becomes (using proper values for a, b, and c)

$$\frac{u^n - u^{n-1}}{\Delta t} = u^n(1 - u^{n-1}), \tag{5}$$

which is a linear algebraic equation in the unknown u^n, making it easy to solve for u^n without any need for an alternative notation.

We shall later refer to the strategy of taking one Picard step, or equivalently, linearizing terms with use of the solution at the previous time step, as the *Picard1* method. It is a widely used approach in science and technology, but with some limitations if Δt is not sufficiently small (as will be illustrated later).

Notice.

Equation (5) does not correspond to a "pure" finite difference method where the equation is sampled at a point and derivatives replaced by differences (because the u^{n-1} term on the right-hand side must then be u^n). The best interpretation of the scheme (5) is a Backward Euler difference combined with a single (perhaps insufficient) Picard iteration at each time level, with the value at the previous time level as start for the Picard iteration.

1.7 Linearization by a geometric mean

We consider now a Crank-Nicolson discretization of (1). This means that the time derivative is approximated by a centered difference,

$$[D_t u = u(1 - u)]^{n+\frac{1}{2}},$$

written out as

$$\frac{u^{n+1}-u^n}{\Delta t} = u^{n+\frac{1}{2}} - (u^{n+\frac{1}{2}})^2\,. \tag{6}$$

The term $u^{n+\frac{1}{2}}$ is normally approximated by an arithmetic mean,

$$u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1}),$$

such that the scheme involves the unknown function only at the time levels where we actually intend to compute it. The same arithmetic mean applied to the nonlinear term gives

$$(u^{n+\frac{1}{2}})^2 \approx \frac{1}{4}(u^n + u^{n+1})^2,$$

which is nonlinear in the unknown u^{n+1}. However, using a *geometric mean* for $(u^{n+\frac{1}{2}})^2$ is a way of linearizing the nonlinear term in (6):

$$(u^{n+\frac{1}{2}})^2 \approx u^n u^{n+1}\,.$$

Using an arithmetic mean on the linear $u^{n+\frac{1}{2}}$ term in (6) and a geometric mean for the second term, results in a linearized equation for the unknown u^{n+1}:

$$\frac{u^{n+1}-u^n}{\Delta t} = \frac{1}{2}(u^n + u^{n+1}) + u^n u^{n+1},$$

which can readily be solved:

$$u^{n+1} = \frac{1 + \frac{1}{2}\Delta t}{1 + \Delta t u^n - \frac{1}{2}\Delta t} u^n\,.$$

This scheme can be coded directly, and since there is no nonlinear algebraic equation to iterate over, we skip the simplified notation with u for u^{n+1} and $u^{(1)}$ for u^n. The technique with using a geometric average is an example of transforming a nonlinear algebraic equation to a linear one, without any need for iterations.

The geometric mean approximation is often very effective for linearizing quadratic nonlinearities. Both the arithmetic and geometric mean approximations have truncation errors of order Δt^2 and are therefore compatible with the truncation error $\mathcal{O}(\Delta t^2)$ of the centered difference approximation for u' in the Crank-Nicolson method.

Applying the operator notation for the means and finite differences, the linearized Crank-Nicolson scheme for the logistic equation can be compactly expressed as

$$[D_t u = \overline{u}^t + \overline{u^2}^{t,g}]^{n+\frac{1}{2}}\,.$$

Remark.

If we use an arithmetic instead of a geometric mean for the nonlinear term in (6), we end up with a nonlinear term $(u^{n+1})^2$. This term can be linearized as $u^- u^{n+1}$ in a Picard iteration approach and in particular as $u^n u^{n+1}$ in a Picard1 iteration approach. The latter gives a scheme almost identical to the one arising from a geometric mean (the difference in u^{n+1} being $\frac{1}{4}\Delta t u^n(u^{n+1} - u^n) \approx \frac{1}{4}\Delta t^2 u' u$, i.e., a difference of size Δt^2).

1.8 Newton's method

The Backward Euler scheme (2) for the logistic equation leads to a nonlinear algebraic equation (3). Now we write any nonlinear algebraic equation in the general and compact form

$$F(u) = 0\,.$$

Newton's method linearizes this equation by approximating $F(u)$ by its Taylor series expansion around a computed value u^- and keeping only the linear part:

$$\begin{aligned} F(u) &= F(u^-) + F'(u^-)(u - u^-) + \frac{1}{2}F''(u^-)(u - u^-)^2 + \cdots \\ &\approx F(u^-) + F'(u^-)(u - u^-) = \hat{F}(u)\,. \end{aligned}$$

The linear equation $\hat{F}(u) = 0$ has the solution

$$u = u^- - \frac{F(u^-)}{F'(u^-)}\,.$$

Expressed with an iteration index in the unknown, Newton's method takes on the more familiar mathematical form

$$u^{k+1} = u^k - \frac{F(u^k)}{F'(u^k)}, \quad k = 0, 1, \ldots$$

It can be shown that the error in iteration $k+1$ of Newton's method is proportional to the square of the error in iteration k, a result referred to as *quadratic convergence*. This means that for small errors the method converges very fast, and in particular much faster than Picard iteration and other iteration methods. (The proof of this result is found in most textbooks on numerical analysis.) However, the quadratic convergence appears only if u^k is sufficiently close to the solution. Further away from the solution the method can easily converge very slowly or diverge. The reader is encouraged to do Exercise 3 to get a better understanding for the behavior of the method.

Application of Newton's method to the logistic equation discretized by the Backward Euler method is straightforward as we have

$$F(u) = au^2 + bu + c, \quad a = \Delta t,\ b = 1 - \Delta t,\ c = -u^{(1)},$$

and then

$$F'(u) = 2au + b\,.$$

The iteration method becomes

$$u = u^- + \frac{a(u^-)^2 + bu^- + c}{2au^- + b}, \quad u^- \ \leftarrow u\,. \tag{7}$$

At each time level, we start the iteration by setting $u^- = u^{(1)}$. Stopping criteria as listed for the Picard iteration can be used also for Newton's method.

An alternative mathematical form, where we write out a, b, and c, and use a time level counter n and an iteration counter k, takes the form

$$u^{n,k+1} = u^{n,k} + \frac{\Delta t(u^{n,k})^2 + (1-\Delta t)u^{n,k} - u^{n-1}}{2\Delta t u^{n,k} + 1 - \Delta t}, \quad u^{n,0} = u^{n-1}, \tag{8}$$

for $k = 0, 1, \ldots$. A program implementation is much closer to (7) than to (8), but the latter is better aligned with the established mathematical notation used in the literature.

1.9 Relaxation

One iteration in Newton's method or Picard iteration consists of solving a linear problem $\hat{F}(u) = 0$. Sometimes convergence problems arise because the new solution u of $\hat{F}(u) = 0$ is "too far away" from the previously computed solution u^-. A remedy is to introduce a relaxation, meaning that we first solve $\hat{F}(u^*) = 0$ for a suggested value u^* and then we take u as a weighted mean of what we had, u^-, and what our linearized equation $\hat{F} = 0$ suggests, u^*:

$$u = \omega u^* + (1-\omega)u^-\,.$$

The parameter ω is known as a *relaxation parameter*, and a choice $\omega < 1$ may prevent divergent iterations.

Relaxation in Newton's method can be directly incorporated in the basic iteration formula:

$$u = u^- - \omega\frac{F(u^-)}{F'(u^-)}\,. \tag{9}$$

1.10 Implementation and experiments

The program `logistic.py`[1] contains implementations of all the methods described above. Below is an extract of the file showing how the Picard and Newton

[1] `http://tinyurl.com/nu656p2/nonlin/logistic.py`

methods are implemented for a Backward Euler discretization of the logistic equation.

```
def BE_logistic(u0, dt, Nt, choice='Picard',
                eps_r=1E-3, omega=1, max_iter=1000):
    if choice == 'Picard1':
        choice = 'Picard'
        max_iter = 1

    u = np.zeros(Nt+1)
    iterations = []
    u[0] = u0
    for n in range(1, Nt+1):
        a = dt
        b = 1 - dt
        c = -u[n-1]

        if choice == 'Picard':

            def F(u):
                return a*u**2 + b*u + c

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = omega*(-c/(a*u_ + b)) + (1-omega)*u_
                k += 1
            u[n] = u_
            iterations.append(k)

        elif choice == 'Newton':

            def F(u):
                return a*u**2 + b*u + c

            def dF(u):
                return 2*a*u + b

            u_ = u[n-1]
            k = 0
            while abs(F(u_)) > eps_r and k < max_iter:
                u_ = u_ - F(u_)/dF(u_)
                k += 1
            u[n] = u_
            iterations.append(k)
    return u, iterations
```

The Crank-Nicolson method utilizing a linearization based on the geometric mean gives a simpler algorithm:

```
def CN_logistic(u0, dt, Nt):
    u = np.zeros(Nt+1)
    u[0] = u0
    for n in range(0, Nt):
        u[n+1] = (1 + 0.5*dt)/(1 + dt*u[n] - 0.5*dt)*u[n]
    return u
```

We may run experiments with the model problem (1) and the different strategies for dealing with nonlinearities as described above. For a quite coarse time resolution, $\Delta t = 0.9$, use of a tolerance $\epsilon_r = 0.1$ in the stopping criterion introduces an iteration error, especially in the Picard iterations, that is visibly much larger than the time discretization error due to a large Δt. This is illustrated by comparing the upper two plots in Figure 1. The one to the right has a stricter tolerance $\epsilon = 10^{-3}$, which causes all the curves corresponding to Picard and Newton iteration to be on top of each other (and no changes can be visually observed by reducing ϵ_r further). The reason why Newton's method does much better than Picard iteration in the upper left plot is that Newton's method with one step comes far below the ϵ_r tolerance, while the Picard iteration needs on average 7 iterations to bring the residual down to $\epsilon_r = 10^{-1}$, which gives insufficient accuracy in the solution of the nonlinear equation. It is obvious that the Picard1 method gives significant errors in addition to the time discretization unless the time step is as small as in the lower right plot.

The *BE exact* curve corresponds to using the exact solution of the quadratic equation at each time level, so this curve is only affected by the Backward Euler time discretization. The *CN gm* curve corresponds to the theoretically more accurate Crank-Nicolson discretization, combined with a geometric mean for linearization. This curve appears more accurate, especially if we take the plot in the lower right with a small Δt and an appropriately small ϵ_r value as the exact curve.

When it comes to the need for iterations, Figure 2 displays the number of iterations required at each time level for Newton's method and Picard iteration. The smaller Δt is, the better starting value we have for the iteration, and the faster the convergence is. With $\Delta t = 0.9$ Picard iteration requires on average 32 iterations per time step, but this number is dramatically reduced as Δt is reduced.

However, introducing relaxation and a parameter $\omega = 0.8$ immediately reduces the average of 32 to 7, indicating that for the large $\Delta t = 0.9$, Picard iteration takes too long steps. An approximately optimal value for ω in this case is 0.5, which results in an average of only 2 iterations! An even more dramatic impact of ω appears when $\Delta t = 1$: Picard iteration does not convergence in 1000 iterations, but $\omega = 0.5$ again brings the average number of iterations down to 2.

Remark. The simple Crank-Nicolson method with a geometric mean for the quadratic nonlinearity gives visually more accurate solutions than the Backward Euler discretization. Even with a tolerance of $\epsilon_r = 10^{-3}$, all the methods for treating the nonlinearities in the Backward Euler discretization give graphs that

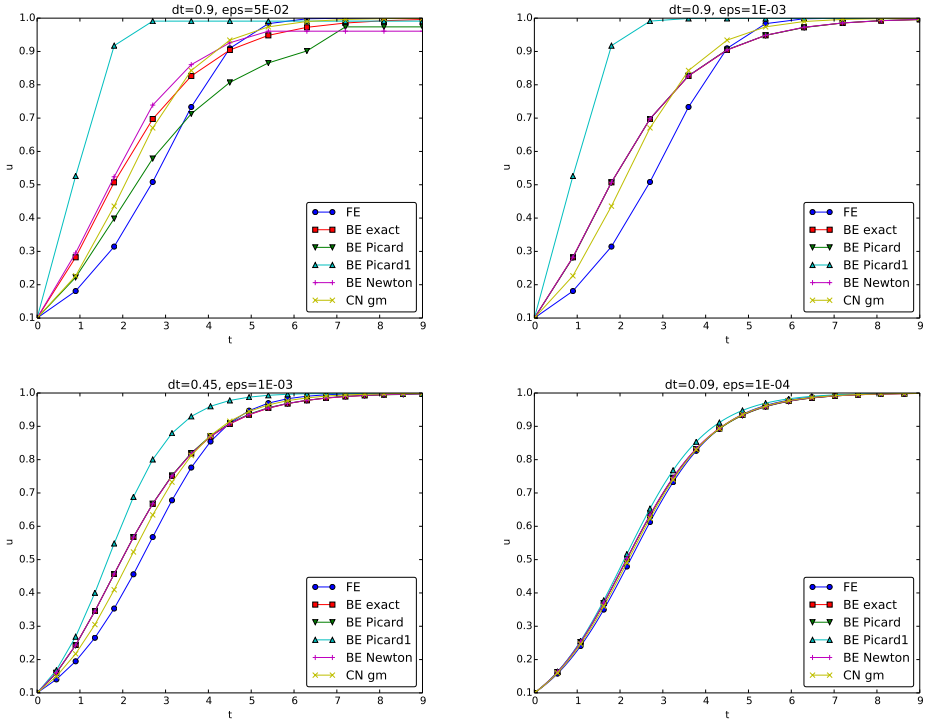

Figure 1: Impact of solution strategy and time step length on the solution.

cannot be distinguished. So for accuracy in this problem, the time discretization is much more crucial than ϵ_r. Ideally, one should estimate the error in the time discretization, as the solution progresses, and set ϵ_r accordingly.

1.11 Generalization to a general nonlinear ODE

Let us see how the various methods in the previous sections can be applied to the more generic model

$$u' = f(u, t), \tag{10}$$

where f is a nonlinear function of u.

Explicit time discretization. Explicit ODE methods like the Forward Euler scheme, Runge-Kutta methods and Adams-Bashforth methods all evaluate f at time levels where u is already computed, so nonlinearities in f do not pose any difficulties.

Backward Euler discretization. Approximating u' by a backward difference leads to a Backward Euler scheme, which can be written as

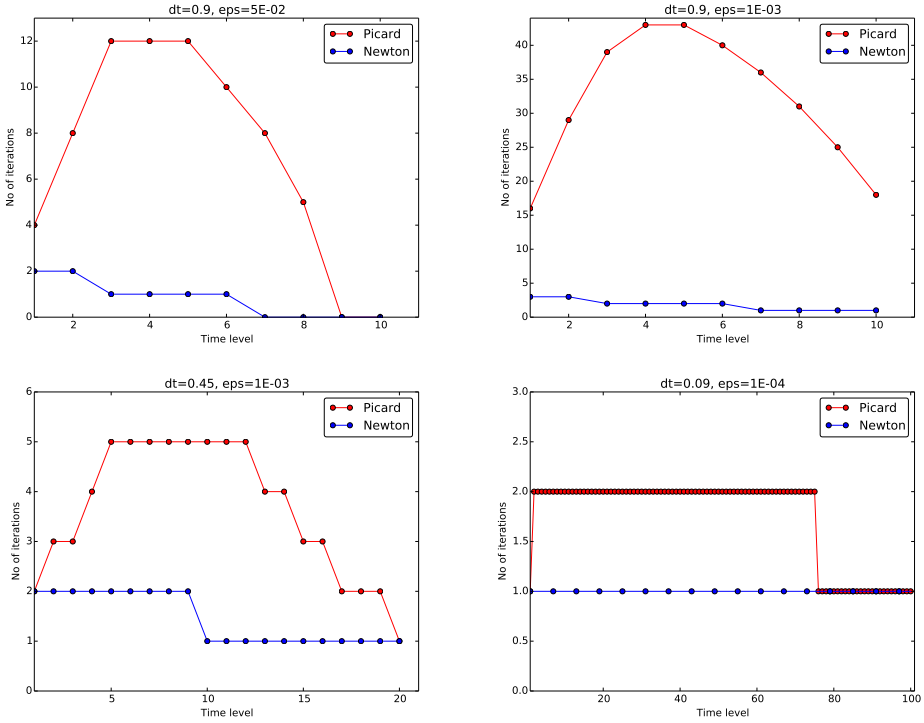

Figure 2: Comparison of the number of iterations at various time levels for Picard and Newton iteration.

$$F(u^n) = u^n - \Delta t\, f(u^n, t_n) - u^{n-1} = 0,$$

or alternatively

$$F(u) = u - \Delta t\, f(u, t_n) - u^{(1)} = 0\,.$$

A simple Picard iteration, not knowing anything about the nonlinear structure of f, must approximate $f(u, t_n)$ by $f(u^-, t_n)$:

$$\hat{F}(u) = u - \Delta t\, f(u^-, t_n) - u^{(1)}\,.$$

The iteration starts with $u^- = u^{(1)}$ and proceeds with repeating

$$u^* = \Delta t\, f(u^-, t_n) + u^{(1)}, \quad u = \omega u^* + (1-\omega)u^-, \quad u^- \leftarrow u,$$

until a stopping criterion is fulfilled.

Explicit vs implicit treatment of nonlinear terms.

Evaluating f for a known u^- is referred to as *explicit* treatment of f, while if $f(u,t)$ has some structure, say $f(u,t) = u^3$, parts of f can involve the unknown u, as in the manual linearization $(u^-)^2u$, and then the treatment of f is "more implicit" and "less explicit". This terminology is inspired by time discretization of $u' = f(u,t)$, where evaluating f for known u values gives explicit schemes, while treating f or parts of f implicitly, makes f contribute to the unknown terms in the equation at the new time level.

Explicit treatment of f usually means stricter conditions on Δt to achieve stability of time discretization schemes. The same applies to iteration techniques for nonlinear algebraic equations: the "less" we linearize f (i.e., the more we keep of u in the original formula), the faster the convergence may be.

We may say that $f(u,t) = u^3$ is treated explicitly if we evaluate f as $(u^-)^3$, partially implicit if we linearize as $(u^-)^2u$ and fully implicit if we represent f by u^3. (Of course, the fully implicit representation will require further linearization, but with $f(u,t) = u^2$ a fully implicit treatment is possible if the resulting quadratic equation is solved with a formula.)

For the ODE $u' = -u^3$ with $f(u,t) = -u^3$ and coarse time resolution $\Delta t = 0.4$, Picard iteration with $(u^-)^2u$ requires 8 iterations with $\epsilon_r = 10^{-3}$ for the first time step, while $(u^-)^3$ leads to 22 iterations. After about 10 time steps both approaches are down to about 2 iterations per time step, but this example shows a potential of treating f more implicitly.

A trick to treat f implicitly in Picard iteration is to evaluate it as $f(u^-,t)u/u^-$. For a polynomial f, $f(u,t) = u^m$, this corresponds to $(u^-)^m u/u^- = (u^-)^{m-1}u$. Sometimes this more implicit treatment has no effect, as with $f(u,t) = \exp(-u)$ and $f(u,t) = \ln(1+u)$, but with $f(u,t) = \sin(2(u+1))$, the $f(u^-,t)u/u^-$ trick leads to 7, 9, and 11 iterations during the first three steps, while $f(u^-,t)$ demands 17, 21, and 20 iterations. (Experiments can be done with the code `ODE_Picard_tricks.py`[a].)

[a] `http://tinyurl.com/nu656p2/nonlin/ODE_Picard_tricks.py`

Newton's method applied to a Backward Euler discretization of $u' = f(u,t)$ requires computation of the derivative

$$F'(u) = 1 - \Delta t \frac{\partial f}{\partial u}(u, t_n)\,.$$

Starting with the solution at the previous time level, $u^- = u^{(1)}$, we can just use the standard formula

$$u = u^- - \omega \frac{F(u^-)}{F'(u^-)} = u^- - \omega \frac{u^- - \Delta t\, f(u^-, t_n) - u^{(1)}}{1 - \Delta t \frac{\partial}{\partial u} f(u^-, t_n)}\,. \tag{11}$$

Crank-Nicolson discretization. The standard Crank-Nicolson scheme with arithmetic mean approximation of f takes the form

$$\frac{u^{n+1}-u^n}{\Delta t} = \frac{1}{2}(f(u^{n+1},t_{n+1}) + f(u^n,t_n))\,.$$

We can write the scheme as a nonlinear algebraic equation

$$F(u) = u - u^{(1)} - \Delta t\frac{1}{2}f(u,t_{n+1}) - \Delta t\frac{1}{2}f(u^{(1)},t_n) = 0\,. \tag{12}$$

A Picard iteration scheme must in general employ the linearization

$$\hat{F}(u) = u - u^{(1)} - \Delta t\frac{1}{2}f(u^-,t_{n+1}) - \Delta t\frac{1}{2}f(u^{(1)},t_n),$$

while Newton's method can apply the general formula (11) with $F(u)$ given in (12) and

$$F'(u) = 1 - \frac{1}{2}\Delta t\frac{\partial f}{\partial u}(u,t_{n+1})\,.$$

1.12 Systems of ODEs

We may write a system of ODEs

$$\begin{aligned}
\frac{d}{dt}u_0(t) &= f_0(u_0(t),u_1(t),\ldots,u_N(t),t),\\
\frac{d}{dt}u_1(t) &= f_1(u_0(t),u_1(t),\ldots,u_N(t),t),\\
&\vdots\\
\frac{d}{dt}u_m(t) &= f_m(u_0(t),u_1(t),\ldots,u_N(t),t),
\end{aligned}$$

as

$$u' = f(u,t), \quad u(0) = U_0, \tag{13}$$

if we interpret u as a vector $u = (u_0(t),u_1(t),\ldots,u_N(t))$ and f as a vector function with components $(f_0(u,t),f_1(u,t),\ldots,f_N(u,t))$.

Most solution methods for scalar ODEs, including the Forward and Backward Euler schemes and the Crank-Nicolson method, generalize in a straightforward way to systems of ODEs simply by using vector arithmetics instead of scalar arithmetics, which corresponds to applying the scalar scheme to each component of the system. For example, here is a backward difference scheme applied to each component,

$$
\begin{aligned}
\frac{u_0^n - u_0^{n-1}}{\Delta t} &= f_0(u^n, t_n), \\
\frac{u_1^n - u_1^{n-1}}{\Delta t} &= f_1(u^n, t_n), \\
&\vdots \\
\frac{u_N^n - u_N^{n-1}}{\Delta t} &= f_N(u^n, t_n),
\end{aligned}
$$

which can be written more compactly in vector form as

$$
\frac{u^n - u^{n-1}}{\Delta t} = f(u^n, t_n)\,.
$$

This is a *system of algebraic equations*,

$$
u^n - \Delta t\, f(u^n, t_n) - u^{n-1} = 0,
$$

or written out

$$
\begin{aligned}
u_0^n - \Delta t\, f_0(u^n, t_n) - u_0^{n-1} &= 0, \\
&\vdots \\
u_N^n - \Delta t\, f_N(u^n, t_n) - u_N^{n-1} &= 0\,.
\end{aligned}
$$

Example. We shall address the 2×2 ODE system for oscillations of a pendulum subject to gravity and air drag. The system can be written as

$$
\dot{\omega} = -\sin\theta - \beta\omega|\omega|, \tag{14}
$$

$$
\dot{\theta} = \omega, \tag{15}
$$

where β is a dimensionless parameter (this is the scaled, dimensionless version of the original, physical model). The unknown components of the system are the angle $\theta(t)$ and the angular velocity $\omega(t)$. We introduce $u_0 = \omega$ and $u_1 = \theta$, which leads to

$$
\begin{aligned}
u_0' &= f_0(u, t) = -\sin u_1 - \beta u_0|u_0|, \\
u_1' &= f_1(u, t) = u_0\,.
\end{aligned}
$$

A Crank-Nicolson scheme reads

$$\frac{u_0^{n+1}-u_0^n}{\Delta t} = -\sin u_1^{n+\frac{1}{2}} - \beta u_0^{n+\frac{1}{2}}|u_0^{n+\frac{1}{2}}|$$
$$\approx -\sin\left(\frac{1}{2}(u_1^{n+1}+u_1 n)\right) - \beta\frac{1}{4}(u_0^{n+1}+u_0^n)|u_0^{n+1}+u_0^n|, \quad (16)$$
$$\frac{u_1^{n+1}-u_1^n}{\Delta t} = u_0^{n+\frac{1}{2}} \approx \frac{1}{2}(u_0^{n+1}+u_0^n)\,. \quad (17)$$

This is a *coupled system* of two nonlinear algebraic equations in two unknowns u_0^{n+1} and u_1^{n+1}.

Using the notation u_0 and u_1 for the unknowns u_0^{n+1} and u_1^{n+1} in this system, writing $u_0^{(1)}$ and $u_1^{(1)}$ for the previous values u_0^n and u_1^n, multiplying by Δt and moving the terms to the left-hand sides, gives

$$u_0 - u_0^{(1)} + \Delta t\,\sin\left(\frac{1}{2}(u_1+u_1^{(1)})\right) + \frac{1}{4}\Delta t\beta(u_0+u_0^{(1)})|u_0+u_0^{(1)}| = 0, \quad (18)$$
$$u_1 - u_1^{(1)} - \frac{1}{2}\Delta t(u_0+u_0^{(1)}) = 0\,. \quad (19)$$

Obviously, we have a need for solving systems of nonlinear algebraic equations, which is the topic of the next section.

2 Systems of nonlinear algebraic equations

Implicit time discretization methods for a system of ODEs, or a PDE, lead to *systems* of nonlinear algebraic equations, written compactly as

$$F(u) = 0,$$

where u is a vector of unknowns $u = (u_0, \ldots, u_N)$, and F is a vector function: $F = (F_0, \ldots, F_N)$. The system at the end of Section 1.12 fits this notation with $N = 1$, $F_0(u)$ given by the left-hand side of (18), while $F_1(u)$ is the left-hand side of (19).

Sometimes the equation system has a special structure because of the underlying problem, e.g.,

$$A(u)u = b(u),$$

with $A(u)$ as an $(N+1)\times(N+1)$ matrix function of u and b as a vector function: $b = (b_0, \ldots, b_N)$.

We shall next explain how Picard iteration and Newton's method can be applied to systems like $F(u) = 0$ and $A(u)u = b(u)$. The exposition has a focus on ideas and practical computations. More theoretical considerations, including quite general results on convergence properties of these methods, can be found in Kelley [1].

2.1 Picard iteration

We cannot apply Picard iteration to nonlinear equations unless there is some special structure. For the commonly arising case $A(u)u = b(u)$ we can linearize the product $A(u)u$ to $A(u^-)u$ and $b(u)$ as $b(u^-)$. That is, we use the most previously computed approximation in A and b to arrive at a *linear system* for u:

$$A(u^-)u = b(u^-)\,.$$

A relaxed iteration takes the form

$$A(u^-)u^* = b(u^-), \quad u = \omega u^* + (1-\omega)u^-\,.$$

In other words, we solve a system of nonlinear algebraic equations as a sequence of linear systems.

Algorithm for relaxed Picard iteration.

Given $A(u)u = b(u)$ and an initial guess u^-, iterate until convergence:

1. solve $A(u^-)u^* = b(u^-)$ with respect to u^*
2. $u = \omega u^* + (1-\omega)u^-$
3. $u^- \leftarrow u$

"Until convergence" means that the iteration is stopped when the change in the unknown, $||u - u^-||$, or the residual $||A(u)u - b||$, is sufficiently small, see Section 2.3 for more details.

2.2 Newton's method

The natural starting point for Newton's method is the general nonlinear vector equation $F(u) = 0$. As for a scalar equation, the idea is to approximate F around a known value u^- by a linear function $\hat{F}$, calculated from the first two terms of a Taylor expansion of F. In the multi-variate case these two terms become

$$F(u^-) + J(u^-)\cdot(u - u^-),$$

where J is the *Jacobian* of F, defined by

$$J_{i,j} = \frac{\partial F_i}{\partial u_j}\,.$$

So, the original nonlinear system is approximated by

$$\hat{F}(u) = F(u^-) + J(u^-) \cdot (u - u^-) = 0,$$

which is linear in u and can be solved in a two-step procedure: first solve $J\delta u = -F(u^-)$ with respect to the vector δu and then update $u = u^- + \delta u$. A relaxation parameter can easily be incorporated:

$$u = \omega(u^- + \delta u) + (1 - \omega)u^- = u^- + \omega\delta u\,.$$

Algorithm for Newton's method.

Given $F(u) = 0$ and an initial guess u^-, iterate until convergence:

1. solve $J\delta u = -F(u^-)$ with respect to δu
2. $u = u^- + \omega\delta u$
3. $u^- \leftarrow u$

For the special system with structure $A(u)u = b(u)$,

$$F_i = \sum_k A_{i,k}(u)u_k - b_i(u),$$

one gets

$$J_{i,j} = \sum_k \frac{\partial A_{i,k}}{\partial u_j} u_k + A_{i,j} - \frac{\partial b_i}{\partial u_j}\,. \tag{20}$$

We realize that the Jacobian needed in Newton's method consists of $A(u^-)$ as in the Picard iteration plus two additional terms arising from the differentiation. Using the notation $A'(u)$ for $\partial A/\partial u$ (a quantity with three indices: $\partial A_{i,k}/\partial u_j$), and $b'(u)$ for $\partial b/\partial u$ (a quantity with two indices: $\partial b_i/\partial u_j$), we can write the linear system to be solved as

$$(A + A'u + b')\delta u = -Au + b,$$

or

$$(A(u^-) + A'(u^-)u^- + b'(u^-))\delta u = -A(u^-)u^- + b(u^-)\,.$$

Rearranging the terms demonstrates the difference from the system solved in each Picard iteration:

$$\underbrace{A(u^-)(u^- + \delta u) - b(u^-)}_{\text{Picard system}} + \gamma(A'(u^-)u^- + b'(u^-))\delta u = 0\,.$$

Here we have inserted a parameter γ such that $\gamma = 0$ gives the Picard system and $\gamma = 1$ gives the Newton system. Such a parameter can be handy in software to easily switch between the methods.

Combined algorithm for Picard and Newton iteration.

Given $A(u)$, $b(u)$, and an initial guess u^-, iterate until convergence:

1. solve $(A+\gamma(A'(u^-)u^- + b'(u^-)))\delta u = -A(u^-)u^- + b(u^-)$ with respect to δu
2. $u = u^- + \omega\delta u$
3. $u^- \leftarrow u$

$\gamma = 1$ gives a Newton method while $\gamma = 0$ corresponds to Picard iteration.

2.3 Stopping criteria

Let $||\cdot||$ be the standard Euclidean vector norm. Four termination criteria are much in use:

- Absolute change in solution: $||u - u^-|| \leq \epsilon_u$
- Relative change in solution: $||u - u^-|| \leq \epsilon_u ||u_0||$, where u_0 denotes the start value of u^- in the iteration
- Absolute residual: $||F(u)|| \leq \epsilon_r$
- Relative residual: $||F(u)|| \leq \epsilon_r ||F(u_0)||$

To prevent divergent iterations to run forever, one terminates the iterations when the current number of iterations k exceeds a maximum value $k_{\max}$.

The relative criteria are most used since they are not sensitive to the characteristic size of u. Nevertheless, the relative criteria can be misleading when the initial start value for the iteration is very close to the solution, since an unnecessary reduction in the error measure is enforced. In such cases the absolute criteria work better. It is common to combine the absolute and relative measures of the size of the residual, as in

$$||F(u)|| \leq \epsilon_{rr}||F(u_0)|| + \epsilon_{ra}, \tag{21}$$

where ϵ_{rr} is the tolerance in the relative criterion and ϵ_{ra} is the tolerance in the absolute criterion. With a very good initial guess for the iteration (typically the solution of a differential equation at the previous time level), the term $||F(u_0)||$ is small and ϵ_{ra} is the dominating tolerance. Otherwise, $\epsilon_{rr}||F(u_0)||$ and the relative criterion dominates.

With the change in solution as criterion we can formulate a combined absolute and relative measure of the change in the solution:

$$||\delta u|| \leq \epsilon_{ur}||u_0|| + \epsilon_{ua}, \tag{22}$$

The ultimate termination criterion, combining the residual and the change in solution with a test on the maximum number of iterations, can be expressed as

$$||F(u)|| \leq \epsilon_{rr}||F(u_0)|| + \epsilon_{ra} \quad \text{or} \quad ||\delta u|| \leq \epsilon_{ur}||u_0|| + \epsilon_{ua} \quad \text{or} \quad k > k_{\max}\,. \tag{23}$$

2.4 Example: A nonlinear ODE model from epidemiology

A very simple model for the spreading of a disease, such as a flu, takes the form of a 2×2 ODE system

$$S' = -\beta SI, \tag{24}$$
$$I' = \beta SI - \nu I, \tag{25}$$

where $S(t)$ is the number of people who can get ill (susceptibles) and $I(t)$ is the number of people who are ill (infected). The constants $\beta > 0$ and $\nu > 0$ must be given along with initial conditions $S(0)$ and $I(0)$.

Implicit time discretization. A Crank-Nicolson scheme leads to a 2×2 system of nonlinear algebraic equations in the unknowns S^{n+1} and I^{n+1}:

$$\frac{S^{n+1} - S^n}{\Delta t} = -\beta[SI]^{n+\frac{1}{2}} \approx -\frac{\beta}{2}(S^n I^n + S^{n+1}I^{n+1}), \tag{26}$$
$$\frac{I^{n+1} - I^n}{\Delta t} = \beta[SI]^{n+\frac{1}{2}} - \nu I^{n+\frac{1}{2}} \approx \frac{\beta}{2}(S^n I^n + S^{n+1}I^{n+1}) - \frac{\nu}{2}(I^n + I^{n+1})\,. \tag{27}$$

Introducing S for S^{n+1}, $S^{(1)}$ for S^n, I for I^{n+1} and $I^{(1)}$ for I^n, we can rewrite the system as

$$F_S(S, I) = S - S^{(1)} + \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + SI) = 0, \tag{28}$$
$$F_I(S, I) = I - I^{(1)} - \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + SI) + \frac{1}{2}\Delta t\nu(I^{(1)} + I) = 0\,. \tag{29}$$

A Picard iteration. We assume that we have approximations S^- and I^- to S and I, respectively. A way of linearizing the only nonlinear term SI is to write I^-S in the $F_S = 0$ equation and S^-I in the $F_I = 0$ equation, which also *decouples* the equations. Solving the resulting linear equations with respect to the unknowns S and I gives

$$S = \frac{S^{(1)} - \frac{1}{2}\Delta t\beta S^{(1)}I^{(1)}}{1 + \frac{1}{2}\Delta t\beta I^-},$$
$$I = \frac{I^{(1)} + \frac{1}{2}\Delta t\beta S^{(1)}I^{(1)} - \frac{1}{2}\Delta t\nu I^{(1)}}{1 - \frac{1}{2}\Delta t\beta S^- + \frac{1}{2}\Delta t\nu}\,.$$

Before a new iteration, we must update $S^- \leftarrow S$ and $I^- \leftarrow I$.

Newton's method. The nonlinear system (28)-(29) can be written as $F(u) = 0$ with $F = (F_S, F_I)$ and $u = (S, I)$. The Jacobian becomes

$$J = \begin{pmatrix} \frac{\partial}{\partial S}F_S & \frac{\partial}{\partial I}F_S \\ \frac{\partial}{\partial S}F_I & \frac{\partial}{\partial I}F_I \end{pmatrix} = \begin{pmatrix} 1 + \frac{1}{2}\Delta t\beta I & \frac{1}{2}\Delta t\beta S \\ -\frac{1}{2}\Delta t\beta I & 1 - \frac{1}{2}\Delta t\beta S + \frac{1}{2}\Delta t\nu \end{pmatrix}.$$

The Newton system $J(u^-)\delta u = -F(u^-)$ to be solved in each iteration is then

$$\begin{pmatrix} 1 + \frac{1}{2}\Delta t\beta I^- & \frac{1}{2}\Delta t\beta S^- \\ -\frac{1}{2}\Delta t\beta I^- & 1 - \frac{1}{2}\Delta t\beta S^- + \frac{1}{2}\Delta t\nu \end{pmatrix} \begin{pmatrix} \delta S \\ \delta I \end{pmatrix} = \begin{pmatrix} S^- - S^{(1)} + \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + S^-I^-) \\ I^- - I^{(1)} - \frac{1}{2}\Delta t\beta(S^{(1)}I^{(1)} + S^-I^-) + \frac{1}{2}\Delta t\nu(I^{(1)} + I^-) \end{pmatrix}$$

Remark. For this particular system of ODEs, explicit time integration methods work very well. Even a Forward Euler scheme is fine, but (as also experienced more generally) the 4-th order Runge-Kutta method is an excellent balance between high accuracy, high efficiency, and simplicity.

3 Linearization at the differential equation level

The attention is now turned to nonlinear partial differential equations (PDEs) and application of the techniques explained above for ODEs. The model problem is a nonlinear diffusion equation for $u(\boldsymbol{x}, t)$:

$$\frac{\partial u}{\partial t} = \nabla\cdot(\alpha(u)\nabla u) + f(u), \qquad \boldsymbol{x}\in\Omega,\ t\in(0,T], \tag{30}$$
$$-\alpha(u)\frac{\partial u}{\partial n} = g, \qquad \boldsymbol{x}\in\partial\Omega_N,\ t\in(0,T], \tag{31}$$
$$u = u_0, \qquad \boldsymbol{x}\in\partial\Omega_D,\ t\in(0,T]\,. \tag{32}$$

In the present section, our aim is to discretize this problem in time and then present techniques for linearizing the time-discrete PDE problem "at the PDE level" such that we transform the nonlinear stationary PDE problem at each time level into a sequence of linear PDE problems, which can be solved using

any method for linear PDEs. This strategy avoids the solution of systems of nonlinear algebraic equations. In Section 4 we shall take the opposite (and more common) approach: discretize the nonlinear problem in time and space first, and then solve the resulting nonlinear algebraic equations at each time level by the methods of Section 2. Very often, the two approaches are mathematically identical, so there is no preference from a computational efficiency point of view. The details of the ideas sketched above will hopefully become clear through the forthcoming examples.

3.1 Explicit time integration

The nonlinearities in the PDE are trivial to deal with if we choose an explicit time integration method for (30), such as the Forward Euler method:

$$[D_t^+ u = \nabla\cdot(\alpha(u)\nabla u) + f(u)]^n,$$

or written out,

$$\frac{u^{n+1}-u^n}{\Delta t} = \nabla\cdot(\alpha(u^n)\nabla u^n) + f(u^n),$$

which is a linear equation in the unknown u^{n+1} with solution

$$u^{n+1} = u^n + \Delta t\nabla\cdot(\alpha(u^n)\nabla u^n) + \Delta t f(u^n)\,.$$

The disadvantage with this discretization is the strict stability criterion $\Delta t \leq h^2/(6\max\alpha)$ for the case $f=0$ and a standard 2nd-order finite difference discretization in 3D space with mesh cell sizes $h=\Delta x=\Delta y=\Delta z$.

3.2 Backward Euler scheme and Picard iteration

A Backward Euler scheme for (30) reads

$$[D_t^- u = \nabla\cdot(\alpha(u)\nabla u) + f(u)]^n\,.$$

Written out,

$$\frac{u^n-u^{n-1}}{\Delta t} = \nabla\cdot(\alpha(u^n)\nabla u^n) + f(u^n)\,. \tag{33}$$

This is a nonlinear PDE for the unknown function $u^n(\boldsymbol{x})$. Such a PDE can be viewed as a time-independent PDE where $u^{n-1}(\boldsymbol{x})$ is a known function.

We introduce a Picard iteration with k as iteration counter. A typical linearization of the $\nabla\cdot(\alpha(u^n)\nabla u^n)$ term in iteration $k+1$ is to use the previously computed $u^{n,k}$ approximation in the diffusion coefficient: $\alpha(u^{n,k})$. The nonlinear source term is treated similarly: $f(u^{n,k})$. The unknown function $u^{n,k+1}$ then fulfills the linear PDE

$$\frac{u^{n,k+1}-u^{n-1}}{\Delta t} = \nabla\cdot(\alpha(u^{n,k})\nabla u^{n,k+1}) + f(u^{n,k})\,. \tag{34}$$

The initial guess for the Picard iteration at this time level can be taken as the solution at the previous time level: $u^{n,0} = u^{n-1}$.

We can alternatively apply the implementation-friendly notation where u corresponds to the unknown we want to solve for, i.e., $u^{n,k+1}$ above, and u^- is the most recently computed value, $u^{n,k}$ above. Moreover, $u^{(1)}$ denotes the unknown function at the previous time level, u^{n-1} above. The PDE to be solved in a Picard iteration then looks like

$$\frac{u - u^{(1)}}{\Delta t} = \nabla \cdot (\alpha(u^-)\nabla u) + f(u^-)\,. \tag{35}$$

At the beginning of the iteration we start with the value from the previous time level: $u^- = u^{(1)}$, and after each iteration, u^- is updated to u.

Remark on notation.

The previous derivations of the numerical scheme for time discretizations of PDEs have, strictly speaking, a somewhat sloppy notation, but it is much used and convenient to read. A more precise notation must distinguish clearly between the exact solution of the PDE problem, here denoted $u_{\mathrm{e}}(\boldsymbol{x}, t)$, and the exact solution of the spatial problem, arising after time discretization at each time level, where (33) is an example. The latter is here represented as $u^n(\boldsymbol{x})$ and is an approximation to $u_{\mathrm{e}}(\boldsymbol{x}, t_n)$. Then we have another approximation $u^{n,k}(\boldsymbol{x})$ to $u^n(\boldsymbol{x})$ when solving the nonlinear PDE problem for u^n by iteration methods, as in (34).

In our notation, u is a synonym for $u^{n,k+1}$ and $u^{(1)}$ is a synonym for u^{n-1}, inspired by what are natural variable names in a code. We will usually state the PDE problem in terms of u and quickly redefine the symbol u to mean the numerical approximation, while u_{e} is not explicitly introduced unless we need to talk about the exact solution and the approximate solution at the same time.

3.3 Backward Euler scheme and Newton's method

At time level n, we have to solve the stationary PDE (33). In the previous section, we saw how this can be done with Picard iterations. Another alternative is to apply the idea of Newton's method in a clever way. Normally, Newton's method is defined for systems of *algebraic equations*, but the idea of the method can be applied at the PDE level too.

Linearization via Taylor expansions. Let $u^{n,k}$ be an approximation to the unknown u^n. We seek a better approximation on the form

$$u^n = u^{n,k} + \delta u\,. \tag{36}$$

The idea is to insert (36) in (33), Taylor expand the nonlinearities and keep only the terms that are linear in δu (which makes (36) an approximation for u^n). Then we can solve a linear PDE for the correction δu and use (36) to find a new approximation

$$u^{n,k+1} = u^{n,k} + \delta u$$

to u^n. Repeating this procedure gives a sequence $u^{n,k+1}$, $k = 0, 1, \ldots$ that hopefully converges to the goal u^n.

Let us carry out all the mathematical details for the nonlinear diffusion PDE discretized by the Backward Euler method. Inserting (36) in (33) gives

$$\frac{u^{n,k} + \delta u - u^{n-1}}{\Delta t} = \nabla \cdot (\alpha(u^{n,k} + \delta u)\nabla(u^{n,k} + \delta u)) + f(u^{n,k} + \delta u)\,. \tag{37}$$

We can Taylor expand $\alpha(u^{n,k} + \delta u)$ and $f(u^{n,k} + \delta u)$:

$$\alpha(u^{n,k} + \delta u) = \alpha(u^{n,k}) + \frac{d\alpha}{du}(u^{n,k})\delta u + \mathcal{O}(\delta u^2) \approx \alpha(u^{n,k}) + \alpha'(u^{n,k})\delta u,$$
$$f(u^{n,k} + \delta u) = f(u^{n,k}) + \frac{df}{du}(u^{n,k})\delta u + \mathcal{O}(\delta u^2) \approx f(u^{n,k}) + f'(u^{n,k})\delta u\,.$$

Inserting the linear approximations of α and f in (37) results in

$$\begin{aligned}\frac{u^{n,k} + \delta u - u^{n-1}}{\Delta t} = {} & \nabla \cdot (\alpha(u^{n,k})\nabla u^{n,k}) + f(u^{n,k}) + \\ & \nabla \cdot (\alpha(u^{n,k})\nabla \delta u) + \nabla \cdot (\alpha'(u^{n,k})\delta u \nabla u^{n,k}) + \\ & \nabla \cdot (\alpha'(u^{n,k})\delta u \nabla \delta u) + f'(u^{n,k})\delta u\,.\end{aligned} \tag{38}$$

The term $\alpha'(u^{n,k})\delta u \nabla \delta u$ is of order δu^2 and therefore omitted since we expect the correction δu to be small ($\delta u \gg \delta u^2$). Reorganizing the equation gives a PDE for δu that we can write in short form as

$$\delta F(\delta u; u^{n,k}) = -F(u^{n,k}),$$

where

$$F(u^{n,k}) = \frac{u^{n,k} - u^{n-1}}{\Delta t} - \nabla \cdot (\alpha(u^{n,k})\nabla u^{n,k}) + f(u^{n,k}), \tag{39}$$

$$\begin{aligned}\delta F(\delta u; u^{n,k}) = {} & -\frac{1}{\Delta t}\delta u + \nabla \cdot (\alpha(u^{n,k})\nabla \delta u) + \\ & \nabla \cdot (\alpha'(u^{n,k})\delta u \nabla u^{n,k}) + f'(u^{n,k})\delta u\,.\end{aligned} \tag{40}$$

Note that δF is a linear function of δu, and F contains only terms that are known, such that the PDE for δu is indeed linear.

Observations.

The notational form $\delta F = -F$ resembles the Newton system $J\delta u = -F$ for systems of algebraic equations, with δF as $J\delta u$. The unknown vector in a linear system of algebraic equations enters the system as a linear operator in terms of a matrix-vector product ($J\delta u$), while at the PDE level we have a linear differential operator instead (δF).

Similarity with Picard iteration. We can rewrite the PDE for δu in a slightly different way too if we define $u^{n,k} + \delta u$ as $u^{n,k+1}$.

$$\frac{u^{n,k+1} - u^{n-1}}{\Delta t} = \nabla\cdot(\alpha(u^{n,k})\nabla u^{n,k+1}) + f(u^{n,k}) + \nabla\cdot(\alpha'(u^{n,k})\delta u\nabla u^{n,k}) + f'(u^{n,k})\delta u\,. \tag{41}$$

Note that the first line is the same PDE as arises in the Picard iteration, while the remaining terms arise from the differentiations that are an inherent ingredient in Newton's method.

Implementation. For coding we want to introduce u for u^n, u^- for $u^{n,k}$ and $u^{(1)}$ for u^{n-1}. The formulas for F and δF are then more clearly written as

$$F(u^-) = \frac{u^- - u^{(1)}}{\Delta t} - \nabla\cdot(\alpha(u^-)\nabla u^-) + f(u^-), \tag{42}$$

$$\delta F(\delta u; u^-) = -\frac{1}{\Delta t}\delta u + \nabla\cdot(\alpha(u^-)\nabla\delta u) + \nabla\cdot(\alpha'(u^-)\delta u\nabla u^-) + f'(u^-)\delta u\,. \tag{43}$$

The form that orders the PDE as the Picard iteration terms plus the Newton method's derivative terms becomes

$$\frac{u - u^{(1)}}{\Delta t} = \nabla\cdot(\alpha(u^-)\nabla u) + f(u^-) + \gamma(\nabla\cdot(\alpha'(u^-)(u-u^-)\nabla u^-) + f'(u^-)(u-u^-))\,. \tag{44}$$

The Picard and full Newton versions correspond to $\gamma = 0$ and $\gamma = 1$, respectively.

Derivation with alternative notation. Some may prefer to derive the linearized PDE for δu using the more compact notation. We start with inserting $u^n = u^- + \delta u$ to get

$$\frac{u^- + \delta u - u^{n-1}}{\Delta t} = \nabla\cdot(\alpha(u^- + \delta u)\nabla(u^- + \delta u)) + f(u^- + \delta u)\,.$$

Taylor expanding,

$$\alpha(u^- + \delta u) \approx \alpha(u^-) + \alpha'(u^-)\delta u,$$
$$f(u^- + \delta u) \approx f(u^-) + f'(u^-)\delta u,$$

and inserting these expressions gives a less cluttered PDE for δu:

$$\frac{u^- + \delta u - u^{n-1}}{\Delta t} = \nabla\cdot(\alpha(u^-)\nabla u^-) + f(u^-) + \nabla\cdot(\alpha(u^-)\nabla\delta u) + \nabla\cdot(\alpha'(u^-)\delta u\nabla u^-) + \nabla\cdot(\alpha'(u^-)\delta u\nabla\delta u) + f'(u^-)\delta u\,.$$

3.4 Crank-Nicolson discretization

A Crank-Nicolson discretization of (30) applies a centered difference at $t_{n+\frac{1}{2}}$:

$$[D_t u = \nabla\cdot(\alpha(u)\nabla u) + f(u)]^{n+\frac{1}{2}}\,.$$

The standard technique is to apply an arithmetic average for quantities defined between two mesh points, e.g.,

$$u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1})\,.$$

However, with nonlinear terms we have many choices of formulating an arithmetic mean:

$$[f(u)]^{n+\frac{1}{2}} \approx f(\frac{1}{2}(u^n + u^{n+1})) = [f(\overline{u}^t)]^{n+\frac{1}{2}}, \tag{45}$$
$$[f(u)]^{n+\frac{1}{2}} \approx \frac{1}{2}(f(u^n) + f(u^{n+1})) = [\overline{f(u)}^t]^{n+\frac{1}{2}}, \tag{46}$$
$$[\alpha(u)\nabla u]^{n+\frac{1}{2}} \approx \alpha(\frac{1}{2}(u^n + u^{n+1}))\nabla(\frac{1}{2}(u^n + u^{n+1})) = [\alpha(\overline{u}^t)\nabla\overline{u}^t]^{n+\frac{1}{2}}, \tag{47}$$
$$[\alpha(u)\nabla u]^{n+\frac{1}{2}} \approx \frac{1}{2}(\alpha(u^n) + \alpha(u^{n+1}))\nabla(\frac{1}{2}(u^n + u^{n+1})) = [\overline{\alpha(u)}^t\nabla\overline{u}^t]^{n+\frac{1}{2}}, \tag{48}$$
$$[\alpha(u)\nabla u]^{n+\frac{1}{2}} \approx \frac{1}{2}(\alpha(u^n)\nabla u^n + \alpha(u^{n+1})\nabla u^{n+1}) = [\overline{\alpha(u)\nabla u}^t]^{n+\frac{1}{2}}\,. \tag{49}$$

A big question is whether there are significant differences in accuracy between taking the products of arithmetic means or taking the arithmetic mean of products. Exercise 6 investigates this question, and the answer is that the approximation is $\mathcal{O}(\Delta t^2)$ in both cases.

4 1D stationary nonlinear differential equations

Section 3 presented methods for linearizing time-discrete PDEs directly prior to discretization in space. We can alternatively carry out the discretization in space of the time-discrete nonlinear PDE problem and get a system of nonlinear algebraic equations, which can be solved by Picard iteration or Newton's method as presented in Section 2. This latter approach will now be described in detail.

We shall work with the 1D problem

$$-(\alpha(u)u')' + au = f(u), \quad x \in (0,L), \quad \alpha(u(0))u'(0) = C,\ u(L) = D\,. \tag{50}$$

The problem (50) arises from the stationary limit of a diffusion equation,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\alpha(u)\frac{\partial u}{\partial x}\right) - au + f(u), \tag{51}$$

as $t \to \infty$ and $\partial u/\partial t \to 0$. Alternatively, the problem (50) arises at each time level from implicit time discretization of (51). For example, a Backward Euler scheme for (51) leads to

$$\frac{u^n - u^{n-1}}{\Delta t} = \frac{d}{dx}\left(\alpha(u^n)\frac{du^n}{dx}\right) - au^n + f(u^n)\,. \tag{52}$$

Introducing $u(x)$ for $u^n(x)$, $u^{(1)}$ for u^{n-1}, and defining $f(u)$ in (50) to be $f(u)$ in (52) plus $u^{n-1}/\Delta t$, gives (50) with $a = 1/\Delta t$.

4.1 Finite difference discretization

The nonlinearity in the differential equation (50) poses no more difficulty than a variable coefficient, as in the term $(\alpha(x)u')'$. We can therefore use a standard finite difference approach when discretizing the Laplace term with a variable coefficient:

$$[-D_x\alpha D_x u + au = f]_i\,.$$

Writing this out for a uniform mesh with points $x_i = i\Delta x$, $i = 0, \ldots, N_x$, leads to

$$-\frac{1}{\Delta x^2}\left(\alpha_{i+\frac{1}{2}}(u_{i+1} - u_i) - \alpha_{i-\frac{1}{2}}(u_i - u_{i-1})\right) + au_i = f(u_i)\,. \tag{53}$$

This equation is valid at all the mesh points $i = 0, 1, \ldots, N_x - 1$. At $i = N_x$ we have the Dirichlet condition $u_i = 0$. The only difference from the case with $(\alpha(x)u')'$ and $f(x)$ is that now α and f are functions of u and not only of x: $(\alpha(u(x))u')'$ and $f(u(x))$.

The quantity $\alpha_{i+\frac{1}{2}}$, evaluated between two mesh points, needs a comment. Since α depends on u and u is only known at the mesh points, we need to express

$\alpha_{i+\frac{1}{2}}$ in terms of u_i and u_{i+1}. For this purpose we use an arithmetic mean, although a harmonic mean is also common in this context if α features large jumps. There are two choices of arithmetic means:

$$\alpha_{i+\frac{1}{2}} \approx \alpha(\frac{1}{2}(u_i + u_{i+1}) = [\alpha(\overline{u}^x)]^{i+\frac{1}{2}}, \tag{54}$$

$$\alpha_{i+\frac{1}{2}} \approx \frac{1}{2}(\alpha(u_i) + \alpha(u_{i+1})) = [\overline{\alpha(u)}^x]^{i+\frac{1}{2}} \tag{55}$$

Equation (53) with the latter approximation then looks like

$$-\frac{1}{2\Delta x^2}\left((\alpha(u_i) + \alpha(u_{i+1}))(u_{i+1} - u_i) - (\alpha(u_{i-1}) + \alpha(u_i))(u_i - u_{i-1})\right) + au_i = f(u_i), \tag{56}$$

or written more compactly,

$$[-D_x\overline{\alpha}^x D_x u + au = f]_i\,.$$

At mesh point $i = 0$ we have the boundary condition $\alpha(u)u' = C$, which is discretized by

$$[\alpha(u)D_{2x}u = C]_0,$$

meaning

$$\alpha(u_0)\frac{u_1 - u_{-1}}{2\Delta x} = C\,. \tag{57}$$

The fictitious value u_{-1} can be eliminated with the aid of (56) for $i = 0$. Formally, (56) should be solved with respect to u_{i-1} and that value (for $i = 0$) should be inserted in (57), but it is algebraically much easier to do it the other way around. Alternatively, one can use a ghost cell $[-\Delta x, 0]$ and update the u_{-1} value in the ghost cell according to (57) after every Picard or Newton iteration. Such an approach means that we use a known u_{-1} value in (56) from the previous iteration.

4.2 Solution of algebraic equations

The structure of the equation system. The nonlinear algebraic equations (56) are of the form $A(u)u = b(u)$ with

$$\begin{aligned}
A_{i,i} &= \frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + 2\alpha(u_i)\alpha(u_{i+1})) + a,\\
A_{i,i-1} &= -\frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + \alpha(u_i)),\\
A_{i,i+1} &= -\frac{1}{2\Delta x^2}(\alpha(u_i) + \alpha(u_{i+1})),\\
b_i &= f(u_i)\,.
\end{aligned}$$

The matrix $A(u)$ is tridiagonal: $A_{i,j} = 0$ for $j > i+1$ and $j < i-1$.

The above expressions are valid for internal mesh points $1 \leq i \leq N_x - 1$. For $i = 0$ we need to express $u_{i-1} = u_{-1}$ in terms of u_1 using (57):

$$u_{-1} = u_1 - \frac{2\Delta x}{\alpha(u_0)} C\,. \tag{58}$$

This value must be inserted in $A_{0,0}$. The expression for $A_{i,i+1}$ applies for $i = 0$, and $A_{i,i-1}$ does not enter the system when $i = 0$.

Regarding the last equation, its form depends on whether we include the Dirichlet condition $u(L) = D$, meaning $u_{N_x} = D$, in the nonlinear algebraic equation system or not. Suppose we choose $(u_0, u_1, \ldots, u_{N_x-1})$ as unknowns, later referred to as *systems without Dirichlet conditions*. The last equation corresponds to $i = N_x - 1$. It involves the boundary value u_{N_x}, which is substituted by D. If the unknown vector includes the boundary value, $(u_0, u_1, \ldots, u_{N_x})$, later referred to as *system including Dirichlet conditions*, the equation for $i = N_x - 1$ just involves the unknown u_{N_x}, and the final equation becomes $u_{N_x} = D$, corresponding to $A_{i,i} = 1$ and $b_i = D$ for $i = N_x$.

Picard iteration. The obvious Picard iteration scheme is to use previously computed values of u_i in $A(u)$ and $b(u)$, as described more in detail in Section 2. With the notation u^- for the most recently computed value of u, we have the system $F(u) \approx \hat{F}(u) = A(u^-)u - b(u^-)$, with $F = (F_0, F_1, \ldots, F_m)$, $u = (u_0, u_1, \ldots, u_m)$. The index m is N_x if the system includes the Dirichlet condition as a separate equation and $N_x - 1$ otherwise. The matrix $A(u^-)$ is tridiagonal, so the solution procedure is to fill a tridiagonal matrix data structure and the right-hand side vector with the right numbers and call a Gaussian elimination routine for tridiagonal linear systems.

Mesh with two cells. It helps on the understanding of the details to write out all the mathematics in a specific case with a small mesh, say just two cells ($N_x = 2$). We use u_i^- for the i-th component in u^-.

The starting point is the basic expressions for the nonlinear equations at mesh point $i = 0$ and $i = 1$:

$$A_{0,-1}u_{-1} + A_{0,0}u_0 + A_{0,1}u_1 = b_0, \tag{59}$$

$$A_{1,0}u_0 + A_{1,1}u_1 + A_{1,2}u_2 = b_1\,. \tag{60}$$

Equation (59) written out reads

$$\frac{1}{2\Delta x^2}(-(\alpha(u_{-1}) + \alpha(u_0))u_{-1} + (\alpha(u_{-1}) + 2\alpha(u_0) + \alpha(u_1))u_0 - (\alpha(u_0) + \alpha(u_1)))u_1 + au_0 = f(u_0)\,.$$

We must then replace u_{-1} by (58). With Picard iteration we get

$$\begin{aligned}\frac{1}{2\Delta x^2}(&-(\alpha(u_{-1}^-)+2\alpha(u_0^-)+\alpha(u_1^-))u_1+\\ &(\alpha(u_{-1}^-)+2\alpha(u_0^-)+\alpha(u_1^-))u_0+au_0\\ &=f(u_0^-)-\frac{1}{\alpha(u_0^-)\Delta x}(\alpha(u_{-1}^-)+\alpha(u_0^-))C,\end{aligned}$$

where

$$u_{-1}^- = u_1^- - \frac{2\Delta x}{\alpha(u_0^-)}C\,.$$

Equation (60) contains the unknown u_2 for which we have a Dirichlet condition. In case we omit the condition as a separate equation, (60) with Picard iteration becomes

$$\begin{aligned}\frac{1}{2\Delta x^2}(&-(\alpha(u_0^-)+\alpha(u_1^-))u_0+\\ &(\alpha(u_0^-)+2\alpha(u_1^-)+\alpha(u_2^-))u_1-\\ &(\alpha(u_1^-)+\alpha(u_2^-)))u_2+au_1=f(u_1^-)\,.\end{aligned}$$

We must now move the u_2 term to the right-hand side and replace all occurrences of u_2 by D:

$$\begin{aligned}\frac{1}{2\Delta x^2}(&-(\alpha(u_0^-)+\alpha(u_1^-))u_0+\\ &(\alpha(u_0^-)+2\alpha(u_1^-)+\alpha(D)))u_1+au_1\\ &=f(u_1^-)+\frac{1}{2\Delta x^2}(\alpha(u_1^-)+\alpha(D))D\,.\end{aligned}$$

The two equations can be written as a 2×2 system:

$$\begin{pmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1}\end{pmatrix}\begin{pmatrix} u_0 \\ u_1\end{pmatrix} = \begin{pmatrix} d_0 \\ d_1\end{pmatrix},$$

where

$$B_{0,0} = \frac{1}{2\Delta x^2}(\alpha(u_{-1}^-) + 2\alpha(u_0^-) + \alpha(u_1^-)) + a, \tag{61}$$

$$B_{0,1} = -\frac{1}{2\Delta x^2}(\alpha(u_{-1}^-) + 2\alpha(u_0^-) + \alpha(u_1^-)), \tag{62}$$

$$B_{1,0} = -\frac{1}{2\Delta x^2}(\alpha(u_0^-) + \alpha(u_1^-)), \tag{63}$$

$$B_{1,1} = \frac{1}{2\Delta x^2}(\alpha(u_0^-) + 2\alpha(u_1^-) + \alpha(D)) + a, \tag{64}$$

$$d_0 = f(u_0^-) - \frac{1}{\alpha(u_0^-)\Delta x}(\alpha(u_{-1}^-) + \alpha(u_0^-))C, \tag{65}$$

$$d_1 = f(u_1^-) + \frac{1}{2\Delta x^2}(\alpha(u_1^-) + \alpha(D))D\,. \tag{66}$$

The system with the Dirichlet condition becomes

$$\begin{pmatrix} B_{0,0} & B_{0,1} & 0 \\ B_{1,0} & B_{1,1} & B_{1,2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ D \end{pmatrix},$$

with

$$B_{1,1} = \frac{1}{2\Delta x^2}(\alpha(u_0^-) + 2\alpha(u_1^-) + \alpha(u_2)) + a, \tag{67}$$

$$B_{1,2} = -\frac{1}{2\Delta x^2}(\alpha(u_1^-) + \alpha(u_2))), \tag{68}$$

$$d_1 = f(u_1^-)\,. \tag{69}$$

Other entries are as in the 2×2 system.

Newton's method. The Jacobian must be derived in order to use Newton's method. Here it means that we need to differentiate $F(u) = A(u)u - b(u)$ with respect to the unknown parameters $u_0, u_1, \ldots, u_m$ ($m = N_x$ or $m = N_x - 1$, depending on whether the Dirichlet condition is included in the nonlinear system $F(u) = 0$ or not). Nonlinear equation number i has the structure

$$F_i = A_{i,i-1}(u_{i-1}, u_i)u_{i-1} + A_{i,i}(u_{i-1}, u_i, u_{i+1})u_i + A_{i,i+1}(u_i, u_{i+1})u_{i+1} - b_i(u_i)\,.$$

Computing the Jacobian requires careful differentiation. For example,

$$\begin{aligned}
\frac{\partial}{\partial u_i}(A_{i,i}(u_{i-1},u_i,u_{i+1})u_i) &= \frac{\partial A_{i,i}}{\partial u_i}u_i + A_{i,i}\frac{\partial u_i}{\partial u_i} \\
&= \frac{\partial}{\partial u_i}(\frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1})) + a)u_i + \\
&\quad \frac{1}{2\Delta x^2}(\alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1})) + a \\
&= \frac{1}{2\Delta x^2}(2\alpha'(u_i)u_i + \alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1})) + a\,.
\end{aligned}$$

The complete Jacobian becomes

$$\begin{aligned}
J_{i,i} = \frac{\partial F_i}{\partial u_i} &= \frac{\partial A_{i,i-1}}{\partial u_i}u_{i-1} + \frac{\partial A_{i,i}}{\partial u_i}u_i + A_{i,i} + \frac{\partial A_{i,i+1}}{\partial u_i}u_{i+1} - \frac{\partial b_i}{\partial u_i} \\
&= \frac{1}{2\Delta x^2}(-\alpha'(u_i)u_{i-1} + 2\alpha'(u_i)u_i + \alpha(u_{i-1}) + 2\alpha(u_i) + \alpha(u_{i+1})) + \\
&\quad a - \frac{1}{2\Delta x^2}\alpha'(u_i)u_{i+1} - b'(u_i), \\
J_{i,i-1} = \frac{\partial F_i}{\partial u_{i-1}} &= \frac{\partial A_{i,i-1}}{\partial u_{i-1}}u_{i-1} + A_{i-1,i} + \frac{\partial A_{i,i}}{\partial u_{i-1}}u_i - \frac{\partial b_i}{\partial u_{i-1}} \\
&= \frac{1}{2\Delta x^2}(-\alpha'(u_{i-1})u_{i-1} - (\alpha(u_{i-1}) + \alpha(u_i)) + \alpha'(u_{i-1})u_i), \\
J_{i,i+1} = \frac{\partial A_{i,i+1}}{\partial u_{i-1}}u_{i+1} &+ A_{i+1,i} + \frac{\partial A_{i,i}}{\partial u_{i+1}}u_i - \frac{\partial b_i}{\partial u_{i+1}} \\
&= \frac{1}{2\Delta x^2}(-\alpha'(u_{i+1})u_{i+1} - (\alpha(u_i) + \alpha(u_{i+1})) + \alpha'(u_{i+1})u_i)\,.
\end{aligned}$$

The explicit expression for nonlinear equation number i, $F_i(u_0, u_1, \ldots)$, arises from moving the $f(u_i)$ term in (56) to the left-hand side:

$$\begin{aligned}
F_i = -\frac{1}{2\Delta x^2}\,&((\alpha(u_i) + \alpha(u_{i+1}))(u_{i+1} - u_i) - (\alpha(u_{i-1}) + \alpha(u_i))(u_i - u_{i-1})) \\
&+ au_i - f(u_i) = 0\,.
\end{aligned} \tag{70}$$

At the boundary point $i = 0$, u_{-1} must be replaced using the formula (58). When the Dirichlet condition at $i = N_x$ is not a part of the equation system, the last equation $F_m = 0$ for $m = N_x - 1$ involves the quantity u_{N_x-1} which must be replaced by D. If u_{N_x} is treated as an unknown in the system, the last equation $F_m = 0$ has $m = N_x$ and reads

$$F_{N_x}(u_0, \ldots, u_{N_x}) = u_{N_x} - D = 0\,.$$

Similar replacement of u_{-1} and u_{N_x} must be done in the Jacobian for the first and last row. When u_{N_x} is included as an unknown, the last row in the Jacobian must help implement the condition $\delta u_{N_x} = 0$, since we assume that u contains

the right Dirichlet value at the beginning of the iteration ($u_{N_x} = D$), and then the Newton update should be zero for $i = 0$, i.e., $\delta u_{N_x} = 0$. This also forces the right-hand side to be $b_i = 0$, $i = N_x$.

We have seen, and can see from the present example, that the linear system in Newton's method contains all the terms present in the system that arises in the Picard iteration method. The extra terms in Newton's method can be multiplied by a factor such that it is easy to program one linear system and set this factor to 0 or 1 to generate the Picard or Newton system.

5 Multi-dimensional nonlinear PDE problems

The fundamental ideas in the derivation of F_i and $J_{i,j}$ in the 1D model problem are easily generalized to multi-dimensional problems. Nevertheless, the expressions involved are slightly different, with derivatives in x replaced by ∇, so we present some examples below in detail.

5.1 Finite difference discretization

A typical diffusion equation

$$u_t = \nabla \cdot (\alpha(u)\nabla u) + f(u),$$

can be discretized by (e.g.) a Backward Euler scheme, which in 2D can be written

$$[D_t^- u = D_x \overline{\alpha(u)}^x D_x u + D_y \overline{\alpha(u)}^y D_y u + f(u)]_{i,j}^n .$$

We do not dive into the details of handling boundary conditions now. Dirichlet and Neumann conditions are handled as in corresponding linear, variable-coefficient diffusion problems.

Writing the scheme out, putting the unknown values on the left-hand side and known values on the right-hand side, and introducing $\Delta x = \Delta y = h$ to save some writing, one gets

$$\begin{aligned} u_{i,j}^n - \frac{\Delta t}{h^2}(&\frac{1}{2}(\alpha(u_{i,j}^n) + \alpha(u_{i+1,j}^n))(u_{i+1,j}^n - u_{i,j}^n) \\ &- \frac{1}{2}(\alpha(u_{i-1,j}^n) + \alpha(u_{i,j}^n))(u_{i,j}^n - u_{i-1,j}^n) \\ &+ \frac{1}{2}(\alpha(u_{i,j}^n) + \alpha(u_{i,j+1}^n))(u_{i,j+1}^n - u_{i,j}^n) \\ &- \frac{1}{2}(\alpha(u_{i,j-1}^n) + \alpha(u_{i,j}^n))(u_{i,j}^n - u_{i-1,j-1}^n)) - \Delta t f(u_{i,j}^n) = u_{i,j}^{n-1} \end{aligned}$$

This defines a nonlinear algebraic system on the form $A(u)u = b(u)$.

Picard iteration. The most recently computed values u^- of u^n can be used in α and f for a Picard iteration, or equivalently, we solve $A(u^-)u = b(u^-)$. The result is a linear system of the same type as arising from $u_t = \nabla\cdot(\alpha(\boldsymbol{x})\nabla u) + f(\boldsymbol{x},t)$.

The Picard iteration scheme can also be expressed in operator notation:

$$[D_t^- u = D_x\overline{\alpha(u^-)}^x D_x u + D_y\overline{\alpha(u^-)}^y D_y u + f(u^-)]_{i,j}^n\,.$$

Newton's method. As always, Newton's method is technically more involved than Picard iteration. We first define the nonlinear algebraic equations to be solved, drop the superscript n (use u for u^n), and introduce $u^{(1)}$ for u^{n-1}:

$$\begin{aligned}F_{i,j} = u_{i,j} - \frac{\Delta t}{h^2}(&\\ &\frac{1}{2}(\alpha(u_{i,j}) + \alpha(u_{i+1,j}))(u_{i+1,j} - u_{i,j}) -\\ &\frac{1}{2}(\alpha(u_{i-1,j}) + \alpha(u_{i,j}))(u_{i,j} - u_{i-1,j}) +\\ &\frac{1}{2}(\alpha(u_{i,j}) + \alpha(u_{i,j+1}))(u_{i,j+1} - u_{i,j}) -\\ &\frac{1}{2}(\alpha(u_{i,j-1}) + \alpha(u_{i,j}))(u_{i,j} - u_{i-1,j-1})) - \Delta t\, f(u_{i,j}) - u^{(1)}_{i,j} = 0\,.\end{aligned}$$

It is convenient to work with two indices i and j in 2D finite difference discretizations, but it complicates the derivation of the Jacobian, which then gets four indices. (Make sure you really understand the 1D version of this problem as treated in Section 4.1.) The left-hand expression of an equation $F_{i,j} = 0$ is to be differentiated with respect to each of the unknowns $u_{r,s}$ (recall that this is short notation for $u^n_{r,s}$), $r \in \mathcal{I}_x$, $s \in \mathcal{I}_y$:

$$J_{i,j,r,s} = \frac{\partial F_{i,j}}{\partial u_{r,s}}\,.$$

The Newton system to be solved in each iteration can be written as

$$\sum_{r\in\mathcal{I}_x}\sum_{s\in\mathcal{I}_y} J_{i,j,r,s}\delta u_{r,s} = -F_{i,j}, \quad i \in \mathcal{I}_x,\ j \in \mathcal{I}_y\,.$$

Given i and j, only a few r and s indices give nonzero contribution to the Jacobian since $F_{i,j}$ contains $u_{i\pm1,j}$, $u_{i,j\pm1}$, and $u_{i,j}$. This means that $J_{i,j,r,s}$ has nonzero contributions only if $r = i \pm 1$, $s = j \pm 1$, as well as $r = i$ and $s = j$. The corresponding terms in $J_{i,j,r,s}$ are $J_{i,j,i-1,j}$, $J_{i,j,i+1,j}$, $J_{i,j,i,j-1}$, $J_{i,j,i,j+1}$ and $J_{i,j,i,j}$. Therefore, the left-hand side of the Newton system, $\sum_r\sum_s J_{i,j,r,s}\delta u_{r,s}$ collapses to

$$\begin{aligned}J_{i,j,r,s}\delta u_{r,s} = J_{i,j,i,j}\delta u_{i,j} + J_{i,j,i-1,j}\delta u_{i-1,j} + J_{i,j,i+1,j}\delta u_{i+1,j} + J_{i,j,i,j-1}\delta u_{i,j-1}&\\ + J_{i,j,i,j+1}\delta u_{i,j+1}&\end{aligned}$$

The specific derivatives become

$$\begin{aligned}
J_{i,j,i-1,j} &= \frac{\partial F_{i,j}}{\partial u_{i-1,j}} \\
&= \frac{\Delta t}{h^2}(\alpha'(u_{i-1,j})(u_{i,j} - u_{i-1,j}) + \alpha(u_{i-1,j})(-1)), \\
J_{i,j,i+1,j} &= \frac{\partial F_{i,j}}{\partial u_{i+1,j}} \\
&= \frac{\Delta t}{h^2}(-\alpha'(u_{i+1,j})(u_{i+1,j} - u_{i,j}) - \alpha(u_{i-1,j})), \\
J_{i,j,i,j-1} &= \frac{\partial F_{i,j}}{\partial u_{i,j-1}} \\
&= \frac{\Delta t}{h^2}(\alpha'(u_{i,j-1})(u_{i,j} - u_{i,j-1}) + \alpha(u_{i,j-1})(-1)), \\
J_{i,j,i,j+1} &= \frac{\partial F_{i,j}}{\partial u_{i,j+1}} \\
&= \frac{\Delta t}{h^2}(-\alpha'(u_{i,j+1})(u_{i,j+1} - u_{i,j}) - \alpha(u_{i,j-1}))\,.
\end{aligned}$$

The $J_{i,j,i,j}$ entry has a few more terms and is left as an exercise. Inserting the most recent approximation u^- for u in the J and F formulas and then forming $J\delta u = -F$ gives the linear system to be solved in each Newton iteration. Boundary conditions will affect the formulas when any of the indices coincide with a boundary value of an index.

5.2 Continuation methods

Picard iteration or Newton's method may diverge when solving PDEs with severe nonlinearities. Relaxation with $\omega < 1$ may help, but in highly nonlinear problems it can be necessary to introduce a *continuation parameter* Λ in the problem: $\Lambda = 0$ gives a version of the problem that is easy to solve, while $\Lambda = 1$ is the target problem. The idea is then to increase Λ in steps, $\Lambda_0 = 0, \Lambda_1 < \cdots < \Lambda_n = 1$, and use the solution from the problem with Λ_{i-1} as initial guess for the iterations in the problem corresponding to Λ_i.

The continuation method is easiest to understand through an example. Suppose we intend to solve

$$-\nabla \cdot (||\nabla u||^q \nabla u) = f,$$

which is an equation modeling the flow of a non-Newtonian fluid through a channel or pipe. For $q = 0$ we have the Poisson equation (corresponding to a Newtonian fluid) and the problem is linear. A typical value for pseudo-plastic fluids may be $q_n = -0.8$. We can introduce the continuation parameter $\Lambda \in [0, 1]$ such that $q = q_n\Lambda$. Let $\{\Lambda_\ell\}_{\ell=0}^n$ be the sequence of Λ values in $[0, 1]$, with corresponding q values $\{q_\ell\}_{\ell=0}^n$. We can then solve a sequence of problems

$$-\nabla\cdot\left(||\nabla u^\ell||_\ell^q\nabla u^\ell\right) = f,\quad \ell = 0,\ldots,n,$$

where the initial guess for iterating on u^ℓ is the previously computed solution $u^{\ell-1}$. If a particular Λ_ℓ leads to convergence problems, one may try a smaller increase in Λ: $\Lambda_* = \frac{1}{2}(\Lambda_{\ell-1} + \Lambda_\ell)$, and repeat halving the step in Λ until convergence is reestablished.

6 Operator splitting methods

Operator splitting is a natural and old idea. When a PDE or system of PDEs contains different terms expressing different physics, it is natural to use different numerical methods for different physical processes. This can optimize and simplify the overall solution process. The idea was especially popularized in the context of the Navier-Stokes equations and reaction-diffusion PDEs. Common names for the technique are *operator splitting*, *fractional step* methods, and *split-step* methods. We shall stick to the former name. In the context of nonlinear differential equations, operator splitting can be used to isolate nonlinear terms and simplify the solution methods.

A related technique, often known as dimensional splitting or alternating direction implicit (ADI) methods, is to split the spatial dimensions and solve a 2D or 3D problem as two or three consecutive 1D problems, but this type of splitting is not to be further considered here.

6.1 Ordinary operator splitting for ODEs

Consider first an ODE where the right-hand side is split into two terms:

$$u' = f_0(u) + f_1(u)\,. \tag{71}$$

In case f_0 and f_1 are linear functions of u, $f_0 = au$ and $f_1 = bu$, we have $u(t) = Ie^{(a+b)t}$, if $u(0) = I$. When going one time step of length Δt from t_n to t_{n+1}, we have

$$u(t_{n+1}) = u(t_n)e^{(a+b)\Delta t}\,.$$

This expression can be also be written as

$$u(t_{n+1}) = u(t_n)e^{a\Delta t}e^{b\Delta t},$$

or

$$u^* = u(t_n)e^{a\Delta t}, \tag{72}$$

$$u(t_{n+1}) = u^*e^{b\Delta t} \tag{73}$$

The first step (72) means solving $u' = f_0$ over a time interval Δt with $u(t_n)$ as start value. The second step (73) means solving $u' = f_1$ over a time interval Δt

with the value at the end of the first step as start value. That is, we progress the solution in two steps and solve two ODEs $u' = f_0$ and $u' = f_1$. The order of the equations is not important. From the derivation above we see that solving $u' = f_1$ prior to $u' = f_0$ can equally well be done.

The technique is exact if the ODEs are linear. For nonlinear ODEs it is only an approximate method with error Δt. The technique can be extended to an arbitrary number of steps; i.e., we may split the PDE system into any number of subsystems. Examples will illuminate this principle.

6.2 Strang splitting for ODEs

The accuracy of the splitting method in Section 6.1 can be improved from $\mathcal{O}(\Delta t)$ to $\mathcal{O}(\Delta t^2)$ using so-called *Strang splitting*, where we take half a step with the f_0 operator, a full step with the f_1 operator, and finally half another step with the f_0 operator. During a time interval Δt the algorithm can be written as follows.

$$
\begin{aligned}
\frac{du^*}{dt} &= f_0(u^*), \quad u^*(t_n) = u(t_n), \quad t \in [t_n, t_n + \frac{1}{2}\Delta t], \\
\frac{du^{***}}{dt} &= f_1(u^{***}), \quad u^{***}(t_n) = u^*(t_{n+\frac{1}{2}}), \quad t \in [t_n, t_n + \Delta t], \\
\frac{du^{**}}{dt} &= f_0(u^{**}), \quad u^{**}(t_{n+\frac{1}{2}}) = u^{***}(t_{n+1}), \quad t \in [t_n + \frac{1}{2}\Delta t, t_n + \Delta t]\,.
\end{aligned}
$$

The global solution is set as $u(t_{n+1}) = u^{**}(t_{n+1})$.

There is no use in combining higher-order methods with ordinary splitting since the error due to splitting is $\mathcal{O}(\Delta t)$, but for Strang splitting it makes sense to use schemes of order $\mathcal{O}(\Delta t^2)$.

With the notation introduced for Strang splitting, we may express ordinary first-order splitting as

$$
\begin{aligned}
\frac{du^*}{dt} &= f_0(u^*), \quad u^*(t_n) = u(t_n), \quad t \in [t_n, t_n + \Delta t], \\
\frac{du^{**}}{dt} &= f_1(u^{**}), \quad u^{**}(t_n) = u^*(t_{n+1}), \quad t \in [t_n, t_n + \Delta t],
\end{aligned}
$$

with global solution set as $u(t_{n+1}) = u^{**}(t_{n+1})$.

6.3 Example: Logistic growth

Let us split the (scaled) logistic equation

$$u' = u(1-u), \quad u(0) = 0.1,$$

with solution $u = (9e^{-t} + 1)^{-1}$, into

$$u' = u - u^2 = f_0(u) + f_1(u), \quad f_0(u) = u, \quad f_1(u) = -u^2\,.$$

We solve $u' = f_0(u)$ and $u' = f_1(u)$ by a Forward Euler step. In addition, we add a method where we solve $u' = f_0(u)$ analytically, since the equation is actually $u' = u$ with solution e^t. The software that accompanies the following methods is the file `split_logistic.py`[2].

Splitting techniques. Ordinary splitting takes a Forward Euler step for each of the ODEs according to

$$\frac{u^{*,n+1} - u^{*,n}}{\Delta t} = f_0(u^{*,n}), \quad u^{*,n} = u(t_n), \quad t \in [t_n, t_n + \Delta t], \tag{74}$$

$$\frac{u^{**,n+1} - u^{**,n}}{\Delta t} = f_1(u^{**,n}), \quad u^{**,n} = u^{*,n+1}, \quad t \in [t_n, t_n + \Delta t], \tag{75}$$

with $u(t_{n+1}) = u^{**,n+1}$.

Strang splitting takes the form

$$\frac{u^{*,n+\frac{1}{2}} - u^{*,n}}{\frac{1}{2}\Delta t} = f_0(u^{*,n}), \quad u^{*,n} = u(t_n), \quad t \in [t_n, t_n + \frac{1}{2}\Delta t], \tag{76}$$

$$\frac{u^{***,n+1} - u^{***,n}}{\Delta t} = f_1(u^{***,n}), \quad u^{***,n} = u^{*,n+\frac{1}{2}}, \quad t \in [t_n, t_n + \Delta t], \tag{77}$$

$$\frac{u^{**,n+1} - u^{**,n+\frac{1}{2}}}{\frac{1}{2}\Delta t} = f_0(u^{**,n+\frac{1}{2}}), \quad u^{**,n+\frac{1}{2}} = u^{***,n+1}, \quad t \in [t_n + \frac{1}{2}\Delta t, t_n + \Delta t]\,. \tag{78}$$

Verbose implementation. The following function computes four solutions arising from the Forward Euler method, ordinary splitting, Strang splitting, as well as Strang splitting with exact treatment of $u' = f_0(u)$:

```
import numpy as np

def solver(dt, T, f, f_0, f_1):
    """
    Solve u'=f by the Forward Euler method and by ordinary and
    Strang splitting: f(u) = f_0(u) + f_1(u).
    """
    Nt = int(round(T/float(dt)))
    t = np.linspace(0, Nt*dt, Nt+1)
    u_FE = np.zeros(len(t))
    u_split1 = np.zeros(len(t))  # 1st-order splitting
    u_split2 = np.zeros(len(t))  # 2nd-order splitting
    u_split3 = np.zeros(len(t))  # 2nd-order splitting w/exact f_0

    # Set initial values
    u_FE[0] = 0.1
```

[2] http://tinyurl.com/nu656p2/nonlin/split_logistic.py

```
    u_split1[0] = 0.1
    u_split2[0] = 0.1
    u_split3[0] = 0.1

    for n in range(len(t)-1):
        # Forward Euler method
        u_FE[n+1] = u_FE[n] + dt*f(u_FE[n])

        # --- Ordinary splitting ---
        # First step
        u_s_n = u_split1[n]
        u_s = u_s_n + dt*f_0(u_s_n)
        # Second step
        u_ss_n = u_s
        u_ss = u_ss_n + dt*f_1(u_ss_n)
        u_split1[n+1] = u_ss

        # --- Strang splitting ---
        # First step
        u_s_n = u_split2[n]
        u_s = u_s_n + dt/2.*f_0(u_s_n)
        # Second step
        u_sss_n = u_s
        u_sss = u_sss_n + dt*f_1(u_sss_n)
        # Third step
        u_ss_n = u_sss
        u_ss = u_ss_n + dt/2.*f_0(u_ss_n)
        u_split2[n+1] = u_ss

        # --- Strang splitting using exact integrator for u'=f_0 ---
        # First step
        u_s_n = u_split3[n]
        u_s = u_s_n*np.exp(dt/2.)  # exact
        # Second step
        u_sss_n = u_s
        u_sss = u_sss_n + dt*f_1(u_sss_n)
        # Third step
        u_ss_n = u_sss
        u_ss = u_ss_n*np.exp(dt/2.)  # exact
        u_split3[n+1] = u_ss

    return u_FE, u_split1, u_split2, u_split3, t
```

Compact implementation. We have used quite many lines for the steps in the splitting methods. Many will prefer to condense the code a bit, as done here:

```
# Ordinary splitting
u_s = u_split1[n] + dt*f_0(u_split1[n])
u_split1[n+1] = u_s + dt*f_1(u_s)
```

```
# Strang splitting
u_s = u_split2[n] + dt/2.*f_0(u_split2[n])
u_sss = u_s + dt*f_1(u_s)
u_split2[n+1] = u_sss + dt/2.*f_0(u_sss)
# Strang splitting using exact integrator for u'=f_0
u_s = u_split3[n]*np.exp(dt/2.)  # exact
u_ss = u_s + dt*f_1(u_s)
u_split3[n+1] = u_ss*np.exp(dt/2.)
```

Results. Figure 3 shows that the impact of splitting is significant. Interestingly, however, the Forward Euler method applied to the entire problem directly is much more accurate than any of the splitting schemes. We also see that Strang splitting is definitely more accurate than ordinary splitting and that it helps a bit to use an exact solution of $u' = f_0(u)$. With a large time step ($\Delta t = 0.2$, left plot in Figure 3), the asymptotic values are off by 20-30%. A more reasonable time step ($\Delta t = 0.05$, right plot in Figure 3) gives better results, but still the asymptotic values are up to 10% wrong.

As technique for solving nonlinear ODEs, we realize that the present case study is not particularly promising, as the Forward Euler method both linearizes the original problem and provides a solution that is much more accurate than any of the splitting techniques. In complicated multi-physics settings, on the other hand, splitting may be the only feasible way to go, and sometimes you really need to apply different numerics to different parts of a PDE problem. But in very simple problems, like the logistic ODE, splitting is just an inferior technique. Still, the logistic ODE is ideal for introducing all the mathematical details and for investigating the behavior.

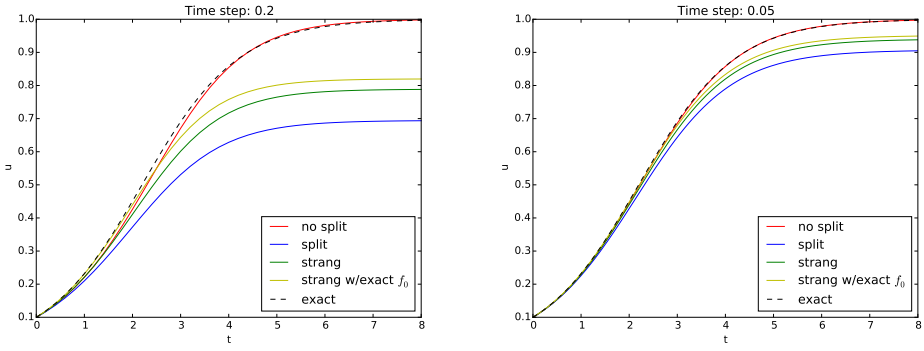

Figure 3: Effect of ordinary and Strang splitting for the logistic equation.

6.4 Reaction-diffusion equation

Consider a diffusion equation coupled to chemical reactions modeled by a nonlinear term $f(u)$:

$$\frac{\partial u}{\partial t} = \alpha\nabla^2 u + f(u)\,.$$

This is a physical process composed of two individual processes: u is the concentration of a substance that is locally generated by a chemical reaction $f(u)$, while u is spreading in space because of diffusion. There are obviously two time scales: one for the chemical reaction and one for diffusion. Typically, fast chemical reactions require much finer time stepping than slower diffusion processes. It could therefore be advantageous to split the two physical effects in separate models and use different numerical methods for the two.

A natural spitting in the present case is

$$\frac{\partial u^*}{\partial t} = \alpha\nabla^2 u^*, \tag{79}$$

$$\frac{\partial u^{**}}{\partial t} = f(u^{**})\,. \tag{80}$$

Looking at these familiar problems, we may apply a θ rule (implicit) scheme for (79) over one time step and avoid dealing with nonlinearities by applying an explicit scheme for (80) over the same time step.

Suppose we have some solution u at time level t_n. For flexibility, we define a θ method for the diffusion part (79) by

$$[D_t u^* = \alpha(D_x D_x u^* + D_y D_y u^*)]^{n+\theta}\,.$$

We use u^n as initial condition for u^*.

The reaction part, which is defined at each mesh point (without coupling values in different mesh points), can employ any scheme for an ODE. Here we use an Adams-Bashforth method of order 2. Recall that the overall accuracy of the splitting method is maximum $\mathcal{O}(\Delta t^2)$ for Strang splitting, otherwise it is just $\mathcal{O}(\Delta t)$. Higher-order methods for ODEs will therefore be a waste of work. The 2nd-order Adams-Bashforth method reads

$$u_{i,j}^{**,n+1} = u_{i,j}^{**,n} + \frac{1}{2}\Delta t\left(3f(u_{i,j}^{**,n}, t_n) - f(u_{i,j}^{**,n-1}, t_{n-1})\right)\,. \tag{81}$$

We can use a Forward Euler step to start the method, i.e, compute $u_{i,j}^{**,1}$.

The algorithm goes like this:

1. Solve the diffusion problem for one time step as usual.
2. Solve the reaction ODEs at each mesh point in $[t_n, t_n + \Delta t]$, using the diffusion solution in 1. as initial condition. The solution of the ODEs constitutes the solution of the original problem at the end of each time step.

We may use a much smaller time step when solving the reaction part, adapted to the dynamics of the problem $u' = f(u)$. This gives great flexibility in splitting methods.

6.5 Example: Reaction-Diffusion with linear reaction term

The methods above may be explored in detail through a specific computational example in which we compute the convergence rates associated with four different solution approaches for the reaction-diffusion equation with a linear reaction term, i.e. $f(u) = -bu$. The methods comprise solving without splitting (just straight Forward Euler), ordinary splitting, first order Strang splitting, and second order Strang splitting. In all four methods, a standard centered difference approximation is used for the spatial second derivative. The methods share the error model $E = Ch^r$, while differing in the step h (being either Δx^2 or Δx) and the convergence rate r (being either 1 or 2).

All code commented below is found in the file `split_diffu_react.py`[3]. When executed, a function `convergence_rates` is called, from which all convergence rate computations are handled:

```
def convergence_rates(scheme='diffusion'):

    F = 0.5     # Upper limit for FE (stability). For CN, this
                # limit does not apply, but for simplicity, we
                # choose F = 0.5 as the initial F value.
    T = 1.2
    a = 3.5
    b = 1
    L = 1.5
    k = np.pi/L

    def exact(x, t):
        '''exact sol. to: du/dt = a*d^2u/dx^2 - b*u'''
        return np.exp(-(a*k**2 + b)*t) * np.sin(k*x)

    def f(u, t):
        return -b*u

    def I(x):
        return exact(x, 0)

    global error    # error computed in the user action function
    error = 0

    # Convergence study
    def action(u, x, t, n):
        global error
        if n == 1:      # New simulation, - reset error
            error = 0
        else:
            error = max(error, np.abs(u - exact(x, t[n])).max())

    E = []
```

[3] `http://tinyurl.com/nu656p2/nonlin/split_diffu_react.py`

```
    h = []
    Nx_values = [10, 20, 40, 80]   # i.e., dx halved each time
    for Nx in Nx_values:
        dx = L/Nx
        if scheme == 'Strang_splitting_2ndOrder':
            print 'Strang splitting with 2nd order schemes...'
            # In this case, E = C*h**r (with r = 2) and since
            # h = dx = K*dt, the ratio dt/dx must be constant.
            # To fulfill this demand, we must let F change
            # when dx changes. From F = a*dt/dx**2, it follows
            # that halving dx AND doubling F assures dt/dx const.
            # Initially, we simply choose F = 0.5.

            dt = F/a*dx**2
            #print 'dt/dx:', dt/dx
            Nt = int(round(T/float(dt)))
            t = np.linspace(0, Nt*dt, Nt+1)   # global time
            Strang_splitting_2ndOrder(I=I, a=a, b=b, f=f, L=L, dt=dt,
                                      dt_Rfactor=1, F=F, t=t, T=T,
                                      user_action=action)
            h.append(dx)
            # prepare for next iteration (make F match dx/2)
            F = F*2       # assures dt/dx const. when dx = dx/2
        else:
            # In these cases, E = C*h**r (with r = 1) and since
            # h = dx**2 = K*dt, the ratio dt/dx**2 must be constant.
            # This is fulfilled by choosing F = 0.5 (for FE stability)
            # and make sure that F, dx and dt comply to F = a*dt/dx**2.
            dt = F/a*dx**2
            Nt = int(round(T/float(dt)))
            t = np.linspace(0, Nt*dt, Nt+1)   # global time
            if scheme == 'diffusion':
                print 'FE on whole eqn...'
                diffusion_theta(I, a, f, L, dt, F, t, T,
                                step_no=0, theta=0,
                                u_L=0, u_R=0, user_action=action)
                h.append(dx**2)
            elif scheme == 'ordinary_splitting':
                print 'Ordinary splitting...'
                ordinary_splitting(I=I, a=a, b=b, f=f, L=L, dt=dt,
                                   dt_Rfactor=1, F=F, t=t, T=T,
                                   user_action=action)
                h.append(dx**2)
            elif scheme == 'Strang_splitting_1stOrder':
                print 'Strang splitting with 1st order schemes...'
                Strang_splitting_1stOrder(I=I, a=a, b=b, f=f, L=L, dt=dt,
                                          dt_Rfactor=1, F=F, t=t, T=T,
                                          user_action=action)
                h.append(dx**2)
            else:
```

```
                print 'Unknown scheme requested!'
                sys.exit(0)

            #print 'dt/dx**2:', dt/dx**2

        E.append(error)
        Nx *= 2         # Nx doubled gives dx/2

    print 'E:', E
    print 'h:', h

    # Convergence rates
    r = [np.log(E[i]/E[i-1])/np.log(h[i]/h[i-1])
         for i in range(1,len(Nx_values))]
    print 'Computed rates:', r


if __name__ == '__main__':

    schemes = ['diffusion',
               'ordinary_splitting',
               'Strang_splitting_1stOrder',
               'Strang_splitting_2ndOrder']

    for scheme in schemes:
        convergence_rates(scheme=scheme)
```

Now, with respect to the error ($E = Ch^r$), the Forward Euler scheme, the ordinary splitting scheme and first order Strang splitting scheme are all first order ($r = 1$), with a step $h = \Delta x^2 = K^{-1}\Delta t$, where K is some constant. This implies that the *ratio* $\frac{\Delta t}{\Delta x^2}$ must be held constant during convergence rate calculations. Furthermore, the Fourier number $F = \frac{\alpha \Delta t}{\Delta x^2}$ is upwards limited to $F = 0.5$, being the stability limit with explicit schemes. Thus, in these cases, we use the fixed value of F and a given (but changing) spatial resolution Δx to compute the corresponding value of Δt according to the expression for F. This assures that $\frac{\Delta t}{\Delta x^2}$ is kept constant. The loop in `convergence_rates` runs over a chosen set of grid points (`Nx_values`) which gives a doubling of spatial resolution with each iteration (Δx is halved).

For the second order Strang splitting scheme, we have $r = 2$ and a step $h = \Delta x = K^{-1}\Delta t$, where K again is some constant. In this case, it is thus the ratio $\frac{\Delta t}{\Delta x}$ that must be held constant during the convergence rate calculations. From the expression for F, it is clear then that F must change with each halving of Δx. In fact, if F is doubled each time Δx is halved, the ratio $\frac{\Delta t}{\Delta x}$ will be constant (this follows, e.g., from the expression for F). This is utilized in our code.

A solver `diffusion_theta` is used in each of the four solution approaches:

```
def diffusion_theta(I, a, f, L, dt, F, t, T, step_no, theta=0.5,
                    u_L=0, u_R=0, user_action=None):
```

```
    """
    Full solver for the model problem using the theta-rule
    difference approximation in time (no restriction on F,
    i.e., the time step when theta >= 0.5).     Vectorized
    implementation and sparse (tridiagonal) coefficient matrix.
    Note that t always covers the whole global time interval, whether
    splitting is the case or not. T, on the other hand, is
    the end of the global time interval if there is no split,
    but if splitting, we use T=dt. When splitting, step_no
    keeps track of the time step number (for lookup in t).
    """

    Nt = int(round(T/float(dt)))
    dx = np.sqrt(a*dt/F)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)       # Mesh points in space
    # Make sure dx and dt are compatible with x and t
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    u   = np.zeros(Nx+1)   # solution array at t[n+1]
    u_1 = np.zeros(Nx+1)   # solution at t[n]

    # Representation of sparse matrix and right-hand side
    diagonal = np.zeros(Nx+1)
    lower    = np.zeros(Nx)
    upper    = np.zeros(Nx)
    b        = np.zeros(Nx+1)

    # Precompute sparse matrix (scipy format)
    Fl = F*theta
    Fr = F*(1-theta)
    diagonal[:] = 1 + 2*Fl
    lower[:] = -Fl  #1
    upper[:] = -Fl  #1
    # Insert boundary conditions
    diagonal[0] = 1
    upper[0] = 0
    diagonal[Nx] = 1
    lower[-1] = 0

    diags = [0, -1, 1]
    A = scipy.sparse.diags(
        diagonals=[diagonal, lower, upper],
        offsets=[0, -1, 1], shape=(Nx+1, Nx+1),
        format='csr')
    #print A.todense()

    # Allow f to be None or 0
    if f is None or f == 0:
```

```
        f = lambda x, t: np.zeros((x.size)) \
            if isinstance(x, np.ndarray) else 0

    # Set initial condition
    if isinstance(I, np.ndarray):   # I is an array
        u_1 = np.copy(I)
    else:                           # I is a function
        for i in range(0, Nx+1):
            u_1[i] = I(x[i])

    if user_action is not None:
        user_action(u_1, x, t, step_no+0)

    # Time loop
    for n in range(0, Nt):
        b[1:-1] = u_1[1:-1] + \
                  Fr*(u_1[:-2] - 2*u_1[1:-1] + u_1[2:]) + \
                  dt*theta*f(u_1[1:-1], t[step_no+n+1]) + \
                  dt*(1-theta)*f(u_1[1:-1], t[step_no+n])
        b[0] = u_L; b[-1] = u_R  # boundary conditions
        u[:] = scipy.sparse.linalg.spsolve(A, b)

        if user_action is not None:
            user_action(u, x, t, step_no+(n+1))

        # Update u_1 before next step
        u_1, u = u, u_1

    # u is now contained in u_1 (swapping)
    return u_1
```

For the no splitting approach with Forward Euler in time, this solver handles both the diffusion and the reaction term. When splitting, `diffusion_theta` takes care of the diffusion term only, while the reaction term is handled either by a Forward Euler scheme in `reaction_FE`, or by a second order Adams-Bashforth scheme from Odespy. The `reaction_FE` function covers one complete time step `dt` during ordinary splitting, while Strang splitting (both first and second order) applies it with `dt/2` twice during each time step `dt`. Since the reaction term typically represents a much faster process than the diffusion term, a further refinement of the time step is made possible in `reaction_FE`. It was implemented as

```
def reaction_FE(I, f, L, Nx, dt, dt_Rfactor, t, step_no,
                user_action=None):
    """Reaction solver, Forward Euler method.
    Note the at t covers the whole global time interval.
    dt is either one complete,or one half, of the step in the
    diffusion part, i.e. there is a local time interval
    [0, dt] or [0, dt/2] that the reaction_FE
    deals with each time it is called. step_no keeps
```

```
    track of the (global) time step number (required
    for lookup in t).
    """

    u = np.copy(I)
    dt_local = dt/float(dt_Rfactor)
    Nt_local = int(round(dt/float(dt_local)))
    x = np.linspace(0, L, Nx+1)

    for n in range(Nt_local):
        time = t[step_no] + n*dt_local
        u[1:Nx] = u[1:Nx] + dt_local*f(u[1:Nx], time)

    # BC already inserted in diffusion step, i.e. no action here
    return u
```

With the ordinary splitting approach, each time step `dt` is covered twice. First computing the impact of the reaction term, then the contribution from the diffusion term:

```
def ordinary_splitting(I, a, b, f, L, dt,
                       dt_Rfactor, F, t, T,
                       user_action=None):
    '''1st order scheme, i.e. Forward Euler is enough for both
    the diffusion and the reaction part. The time step dt is
    given for the diffusion step, while the time step for the
    reaction part is found as dt/dt_Rfactor, where dt_Rfactor >= 1.
    '''
    Nt = int(round(T/float(dt)))
    dx = np.sqrt(a*dt/F)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)       # Mesh points in space
    u = np.zeros(Nx+1)

    # Set initial condition u(x,0) = I(x)
    for i in range(0, Nx+1):
        u[i] = I(x[i])

    # In the following loop, each time step is "covered twice",
    # first for reaction, then for diffusion
    for n in range(0, Nt):
        # Reaction step (potentially many smaller steps within dt)
        u_s = reaction_FE(I=u, f=f, L=L, Nx=Nx,
                          dt=dt, dt_Rfactor=dt_Rfactor,
                          t=t, step_no=n,
                          user_action=None)

        u = diffusion_theta(I=u_s, a=a, f=0, L=L, dt=dt, F=F,
                            t=t, T=dt, step_no=n, theta=0,
                            u_L=0, u_R=0, user_action=None)
```

```
        if user_action is not None:
            user_action(u, x, t, n+1)

    return
```

For the two Strang splitting approaches, each time step `dt` is handled by first computing the reaction step for (the first) `dt/2`, followed by a diffusion step `dt`, before the reaction step is treated once again for (the remaining) `dt/2`. Since first order Strang splitting is no better than first order accurate, both the reaction and diffusion steps are computed explicitly. The solver was implemented as

```
def Strang_splitting_1stOrder(I, a, b, f, L, dt, dt_Rfactor,
                              F, t, T, user_action=None):
    '''Strang splitting while still using FE for the reaction
    step and for the diffusion step. Gives 1st order scheme.
    The time step dt is given for the diffusion step, while
    the time step for the reaction part is found as
    0.5*dt/dt_Rfactor, where dt_Rfactor >= 1. Introduce an
    extra time mesh t2 for the reaction part, since it steps dt/2.
    '''
    Nt = int(round(T/float(dt)))
    t2 = np.linspace(0, Nt*dt, (Nt+1)+Nt)   # Mesh points in diff
    dx = np.sqrt(a*dt/F)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)
    u = np.zeros(Nx+1)

    # Set initial condition u(x,0) = I(x)
    for i in range(0, Nx+1):
        u[i] = I(x[i])

    for n in range(0, Nt):
        # Reaction step (1/2 dt: from t_n to t_n+1/2)
        # (potentially many smaller steps within dt/2)
        u_s = reaction_FE(I=u, f=f, L=L, Nx=Nx,
                          dt=dt/2.0, dt_Rfactor=dt_Rfactor,
                          t=t2, step_no=2*n,
                          user_action=None)
        # Diffusion step (1 dt: from t_n to t_n+1)
        u_sss = diffusion_theta(I=u_s, a=a, f=0, L=L, dt=dt, F=F,
                                t=t, T=dt, step_no=n, theta=0,
                                u_L=0, u_R=0, user_action=None)
        # Reaction step (1/2 dt: from t_n+1/2 to t_n+1)
        # (potentially many smaller steps within dt/2)
        u = reaction_FE(I=u_sss, f=f, L=L, Nx=Nx,
                          dt=dt/2.0, dt_Rfactor=dt_Rfactor,
                          t=t2, step_no=2*n+1,
                          user_action=None)
```

```
        if user_action is not None:
            user_action(u, x, t, n+1)

    return
```

The second order version of the Strang splitting approach utilizes a second order Adams-Bashforth solver for the reaction part and a Crank-Nicolson scheme for the diffusion part. The solver has the same structure as the one for first order Strang splitting and was implemented as

```
def Strang_splitting_2ndOrder(I, a, b, f, L, dt, dt_Rfactor,
                              F, t, T, user_action=None):
    '''Strang splitting using Crank-Nicolson for the diffusion
    step (theta-rule) and Adams-Bashforth 2 for the reaction step.
    Gives 2nd order scheme. Introduce an extra time mesh t2 for
    the reaction part, since it steps dt/2.
    '''
    import odespy
    Nt = int(round(T/float(dt)))
    t2 = np.linspace(0, Nt*dt, (Nt+1)+Nt)   # Mesh points in diff
    dx = np.sqrt(a*dt/F)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)
    u = np.zeros(Nx+1)

    # Set initial condition u(x,0) = I(x)
    for i in range(0, Nx+1):
        u[i] = I(x[i])

    reaction_solver = odespy.AdamsBashforth2(f)

    for n in range(0, Nt):
        # Reaction step (1/2 dt: from t_n to t_n+1/2)
        # (potentially many smaller steps within dt/2)
        reaction_solver.set_initial_condition(u)
        t_points = np.linspace(0, dt/2.0, dt_Rfactor+1)
        u_AB2, t_ = reaction_solver.solve(t_points) # t_ not needed
        u_s = u_AB2[-1,:]  # pick sol at last point in time

        # Diffusion step (1 dt: from t_n to t_n+1)
        u_sss = diffusion_theta(I=u_s, a=a, f=0, L=L, dt=dt, F=F,
                                t=t, T=dt, step_no=n, theta=0.5,
                                u_L=0, u_R=0, user_action=None)
        # Reaction step (1/2 dt: from t_n+1/2 to t_n+1)
        # (potentially many smaller steps within dt/2)
        reaction_solver.set_initial_condition(u_sss)
        t_points = np.linspace(0, dt/2.0, dt_Rfactor+1)
        u_AB2, t_ = reaction_solver.solve(t_points) # t_ not needed
        u = u_AB2[-1,:]  # pick sol at last point in time
```

```
        if user_action is not None:
            user_action(u, x, t, n+1)

    return
```

When executing `split_diffu_react.py`, we find that the estimated convergence rates are as expected. The second order Strang splitting gives the least error (about $4e^{-5}$) and has second order convergence ($r = 2$), while the remaining three approaches have first order convergence ($r = 1$).

6.6 Analysis of the splitting method

Let us address a linear PDE problem for which we can develop analytical solutions of the discrete equations, with and without splitting, and discuss these. Choosing $f(u) = -\beta u$ for a constant β gives a linear problem. We use the Forward Euler method for both the PDE and ODE problems.

We seek a 1D Fourier wave component solution of the problem, assuming homogeneous Dirichlet conditions at $x = 0$ and $x = L$:

$$u = e^{-\alpha k^2 t - \beta t} \sin kx, \quad k = \frac{\pi}{L}\,.$$

This component fits the 1D PDE problem ($f = 0$). On complex form we can write

$$u = e^{-\alpha k^2 t - \beta t + ikx},$$

where $i = \sqrt{-1}$ and the imaginary part is taken as the physical solution.

We refer to Section **??** in [3] and to the book [2] for a discussion of exact numerical solutions to diffusion and decay problems, respectively. The key idea is to search for solutions $A^n e^{ikx}$ and determine A. For the diffusion problem solved by a Forward Euler method one has

$$A = 1 - 4F \sin^p,$$

where $F = \alpha\Delta t/\Delta x^2$ is the mesh Fourier number and $p = k\Delta x/2$ is a dimensionless number reflecting the spatial resolution (number of points per wave length in space). For the decay problem $u' = -\beta u$, we have $A = 1 - q$, where q is a dimensionless parameter for the resolution in the decay problem: $q = \beta\Delta t$.

The original model problem can also be discretized by a Forward Euler scheme,

$$[D_t^+ u = \alpha D_x D_x u - \beta u]_i^n\,.$$

Assuming $A^n e^{ikx}$ we find that

$$u_i^n = (1 - 4F \sin^p - q)^n \sin kx\,.$$

We are particularly interested in what happens at one time step. That is,

$$u_i^n = (1 - 4F\sin^2 p)u_i^{n-1}\,.$$

In the two stage algorithm, we first compute the diffusion step

$$u_i^{*,n+1} = (1 - 4F\sin^2 p)u_i^{n-1}\,.$$

Then we use this as input to the decay algorithm and arrive at

$$u^{**,n+1} = (1-q)u^{*,n+1} = (1-q)(1 - 4F\sin^2 p)u_i^{n-1}\,.$$

The splitting approximation over one step is therefore

$$E = 1 - 4F\sin^p - q - (1-q)(1 - 4F\sin^2 p) = -q(2 - F\sin^2 p))\,.$$

7 Exercises

Problem 1: Determine if equations are nonlinear or not

Classify each term in the following equations as linear or nonlinear. Assume that u, $\boldsymbol{u}$, and p are unknown functions and that all other symbols are known quantities.

1. $mu'' + \beta|u'|u' + cu = F(t)$
2. $u_t = \alpha u_{xx}$
3. $u_{tt} = c^2\nabla^2 u$
4. $u_t = \nabla\cdot(\alpha(u)\nabla u) + f(x,y)$
5. $u_t + f(u)_x = 0$
6. $\boldsymbol{u}_t + \boldsymbol{u}\cdot\nabla\boldsymbol{u} = -\nabla p + r\nabla^2\boldsymbol{u}$, $\nabla\cdot\boldsymbol{u} = 0$ ($\boldsymbol{u}$ is a vector field)
7. $u' = f(u,t)$
8. $\nabla^2 u = \lambda e^u$

Filename: `nonlinear_vs_linear`.

Problem 2: Derive and investigate a generalized logistic model

The logistic model for population growth is derived by assuming a nonlinear growth rate,

$$u' = a(u)u, \quad u(0) = I, \tag{82}$$

and the logistic model arises from the simplest possible choice of $a(u)$: $r(u) = \varrho(1 - u/M)$, where M is the maximum value of u that the environment can sustain, and ϱ is the growth under unlimited access to resources (as in the beginning when u is small). The idea is that $a(u) \sim \varrho$ when u is small and that $a(t) \to 0$ as $u \to M$.

An $a(u)$ that generalizes the linear choice is the polynomial form

$$a(u) = \varrho(1 - u/M)^p, \tag{83}$$

where $p > 0$ is some real number.

a) Formulate a Forward Euler, Backward Euler, and a Crank-Nicolson scheme for (82).

Hint. Use a geometric mean approximation in the Crank-Nicolson scheme: $[a(u)u]^{n+1/2} \approx a(u^n)u^{n+1}$.

b) Formulate Picard and Newton iteration for the Backward Euler scheme in a).

c) Implement the numerical solution methods from a) and b). Use `logistic.py`[4] to compare the case $p = 1$ and the choice (83).

d) Implement unit tests that check the asymptotic limit of the solutions: $u \to M$ as $t \to \infty$.

Hint. You need to experiment to find what "infinite time" is (increases substantially with p) and what the appropriate tolerance is for testing the asymptotic limit.

e) Perform experiments with Newton and Picard iteration for the model (83). See how sensitive the number of iterations is to Δt and p.
Filename: `logistic_p`.

Problem 3: Experience the behavior of Newton's method

The program `Newton_demo.py`[5] illustrates graphically each step in Newton's method and is run like

Terminal

```
Terminal> python Newton_demo.py f dfdx x0 xmin xmax
```

Use this program to investigate potential problems with Newton's method when solving $e^{-0.5x^2}\cos(\pi x) = 0$. Try a starting point $x_0 = 0.8$ and $x_0 = 0.85$ and watch the different behavior. Just run

[4] `http://tinyurl.com/nu656p2/nonlin/logistic.py`
[5] `http://tinyurl.com/nu656p2/nonlin/Newton_demo.py`

Terminal

```
Terminal>  python Newton_demo.py '0.2 + exp(-0.5*x**2)*cos(pi*x)' \
           '-x*exp(-x**2)*cos(pi*x) - pi*exp(-x**2)*sin(pi*x)' \
           0.85 -3 3
```

and repeat with 0.85 replaced by 0.8.

Exercise 4: Compute the Jacobian of a 2×2 system

Write up the system (18)-(19) in the form $F(u) = 0$, $F = (F_0, F_1)$, $u = (u_0, u_1)$, and compute the Jacobian $J_{i,j} = \partial F_i/\partial u_j$.

Problem 5: Solve nonlinear equations arising from a vibration ODE

Consider a nonlinear vibration problem

$$mu'' + bu'|u'| + s(u) = F(t), \tag{84}$$

where $m > 0$ is a constant, $b \geq 0$ is a constant, $s(u)$ a possibly nonlinear function of u, and $F(t)$ is a prescribed function. Such models arise from Newton's second law of motion in mechanical vibration problems where $s(u)$ is a spring or restoring force, mu'' is mass times acceleration, and $bu'|u'|$ models water or air drag.

a) Rewrite the equation for u as a system of two first-order ODEs, and discretize this system by a Crank-Nicolson (centered difference) method. With $v = u'$, we get a nonlinear term $v^{n+\frac{1}{2}}|v^{n+\frac{1}{2}}|$. Use a geometric average for $v^{n+\frac{1}{2}}$.

b) Formulate a Picard iteration method to solve the system of nonlinear algebraic equations.

c) Explain how to apply Newton's method to solve the nonlinear equations at each time level. Derive expressions for the Jacobian and the right-hand side in each Newton iteration.
Filename: `nonlin_vib`.

Exercise 6: Find the truncation error of arithmetic mean of products

In Section 3.4 we introduce alternative arithmetic means of a product. Say the product is $P(t)Q(t)$ evaluated at $t = t_{n+\frac{1}{2}}$. The exact value is

$$[PQ]^{n+\frac{1}{2}} = P^{n+\frac{1}{2}}Q^{n+\frac{1}{2}}$$

There are two obvious candidates for evaluating $[PQ]^{n+\frac{1}{2}}$ as a mean of values of P and Q at t_n and t_{n+1}. Either we can take the arithmetic mean of each factor P and Q,

$$[PQ]^{n+\frac{1}{2}} \approx \frac{1}{2}(P^n + P^{n+1})\frac{1}{2}(Q^n + Q^{n+1}), \tag{85}$$

or we can take the arithmetic mean of the product PQ:

$$[PQ]^{n+\frac{1}{2}} \approx \frac{1}{2}(P^nQ^n + P^{n+1}Q^{n+1})\,. \tag{86}$$

The arithmetic average of $P(t_{n+\frac{1}{2}})$ is $\mathcal{O}(\Delta t^2)$:

$$P(t_{n+\frac{1}{2}}) = \frac{1}{2}(P^n + P^{n+1}) + \mathcal{O}(\Delta t^2)\,.$$

A fundamental question is whether (85) and (86) have different orders of accuracy in $\Delta t = t_{n+1} - t_n$. To investigate this question, expand quantities at t_{n+1} and t_n in Taylor series around $t_{n+\frac{1}{2}}$, and subtract the true value $[PQ]^{n+\frac{1}{2}}$ from the approximations (85) and (86) to see what the order of the error terms are.

Hint. You may explore `sympy` for carrying out the tedious calculations. A general Taylor series expansion of $P(t + \frac{1}{2}\Delta t)$ around t involving just a general function $P(t)$ can be created as follows:

```
>>> from sympy import *
>>> t, dt = symbols('t dt')
>>> P = symbols('P', cls=Function)
>>> P(t).series(t, 0, 4)
P(0) + t*Subs(Derivative(P(_x), _x), (_x,), (0,)) +
t**2*Subs(Derivative(P(_x), _x, _x), (_x,), (0,))/2 +
t**3*Subs(Derivative(P(_x), _x, _x, _x), (_x,), (0,))/6 + O(t**4)
>>> P_p = P(t).series(t, 0, 4).subs(t, dt/2)
>>> P_p
P(0) + dt*Subs(Derivative(P(_x), _x), (_x,), (0,))/2 +
dt**2*Subs(Derivative(P(_x), _x, _x), (_x,), (0,))/8 +
dt**3*Subs(Derivative(P(_x), _x, _x, _x), (_x,), (0,))/48 + O(dt**4)
```

The error of the arithmetic mean, $\frac{1}{2}(P(-\frac{1}{2}\Delta t) + P(-\frac{1}{2}\Delta t))$ for $t = 0$ is then

```
>>> P_m = P(t).series(t, 0, 4).subs(t, -dt/2)
>>> mean = Rational(1,2)*(P_m + P_p)
>>> error = simplify(expand(mean) - P(0))
>>> error
dt**2*Subs(Derivative(P(_x), _x, _x), (_x,), (0,))/8 + O(dt**4)
```

Use these examples to investigate the error of (85) and (86) for $n = 0$. (Choosing $n = 0$ is necessary for not making the expressions too complicated for `sympy`, but there is of course no lack of generality by using $n = 0$ rather than an arbitrary n - the main point is the product and addition of Taylor series.)
Filename: `product_arith_mean`.

Problem 7: Newton's method for linear problems

Suppose we have a linear system $F(u) = Au - b = 0$. Apply Newton's method to this system, and show that the method converges in one iteration. Filename: `Newton_linear`.

Problem 8: Discretize a 1D problem with a nonlinear coefficient

We consider the problem

$$((1 + u^2)u')' = 1, \quad x \in (0, 1), \quad u(0) = u(1) = 0\,. \tag{87}$$

Discretize (87) by a centered finite difference method on a uniform mesh. Filename: `nonlin_1D_coeff_discretize`.

Problem 9: Linearize a 1D problem with a nonlinear coefficient

We have a two-point boundary value problem

$$((1 + u^2)u')' = 1, \quad x \in (0, 1), \quad u(0) = u(1) = 0\,. \tag{88}$$

a) Construct a Picard iteration method for (88) without discretizing in space.

b) Apply Newton's method to (88) without discretizing in space.

c) Discretize (88) by a centered finite difference scheme. Construct a Picard method for the resulting system of nonlinear algebraic equations.

d) Discretize (88) by a centered finite difference scheme. Define the system of nonlinear algebraic equations, calculate the Jacobian, and set up Newton's method for solving the system.
Filename: `nonlin_1D_coeff_linearize`.

Problem 10: Finite differences for the 1D Bratu problem

We address the so-called Bratu problem

$$u'' + \lambda e^u = 0, \quad x \in (0, 1), \quad u(0) = u(1) = 0, \tag{89}$$

where λ is a given parameter and u is a function of x. This is a widely used model problem for studying numerical methods for nonlinear differential equations. The problem (89) has an exact solution

$$u_{\mathrm{e}}(x) = -2\ln\left(\frac{\cosh((x - \frac{1}{2})\theta/2)}{\cosh(\theta/4)}\right),$$

where θ solves

$$\theta = \sqrt{2\lambda}\cosh(\theta/4)\,.$$

There are two solutions of (89) for $0 < \lambda < \lambda_c$ and no solution for $\lambda > \lambda_c$. For $\lambda = \lambda_c$ there is one unique solution. The critical value λ_c solves

$$1 = \sqrt{2\lambda_c}\frac{1}{4}\sinh(\theta(\lambda_c)/4)\,.$$

A numerical value is $\lambda_c = 3.513830719$.

a) Discretize (89) by a centered finite difference method.

b) Set up the nonlinear equations $F_i(u_0, u_1, \ldots, u_{N_x}) = 0$ from a). Calculate the associated Jacobian.

c) Implement a solver that can compute $u(x)$ using Newton's method. Plot the error as a function of x in each iteration.

d) Investigate whether Newton's method gives second-order convergence by computing $||u_{\mathrm{e}} - u||/||u_{\mathrm{e}} - u^-||^2$ in each iteration, where u is solution in the current iteration and u^- is the solution in the previous iteration.
Filename: `nonlin_1D_Bratu_fd`.

Problem 11: Discretize a nonlinear 1D heat conduction PDE by finite differences

We address the 1D heat conduction PDE

$$\varrho c(T)T_t = (k(T)T_x)_x,$$

for $x \in [0, L]$, where ϱ is the density of the solid material, $c(T)$ is the heat capacity, T is the temperature, and $k(T)$ is the heat conduction coefficient. $T(x, 0) = I(x)$, and ends are subject to a cooling law:

$$k(T)T_x|_{x=0} = h(T)(T - T_s), \quad -k(T)T_x|_{x=L} = h(T)(T - T_s),$$

where $h(T)$ is a heat transfer coefficient and T_s is the given surrounding temperature.

a) Discretize this PDE in time using either a Backward Euler or Crank-Nicolson scheme.

b) Formulate a Picard iteration method for the time-discrete problem (i.e., an iteration method before discretizing in space).

c) Formulate a Newton method for the time-discrete problem in b).

d) Discretize the PDE by a finite difference method in space. Derive the matrix and right-hand side of a Picard iteration method applied to the space-time discretized PDE.

e) Derive the matrix and right-hand side of a Newton method applied to the discretized PDE in d).
Filename: `nonlin_1D_heat_FD`.

Problem 12: Differentiate a highly nonlinear term

The operator $\nabla \cdot (\alpha(u)\nabla u)$ with $\alpha(u) = |\nabla u|^q$ appears in several physical problems, especially flow of Non-Newtonian fluids. The expression $|\nabla u|$ is defined as the Euclidean norm of a vector: $|\nabla u|^2 = \nabla u \cdot \nabla u$. In a Newton method one has to carry out the differentiation $\partial\alpha(u)/\partial c_j$, for $u = \sum_k c_k\psi_k$. Show that

$$\frac{\partial}{\partial u_j}|\nabla u|^q = q|\nabla u|^{q-2}\nabla u \cdot \nabla\psi_j \,.$$

Filename: `nonlin_differentiate`.

Exercise 13: Crank-Nicolson for a nonlinear 3D diffusion equation

Redo Section 5.1 when a Crank-Nicolson scheme is used to discretize the equations in time and the problem is formulated for three spatial dimensions.

Hint. Express the Jacobian as $J_{i,j,k,r,s,t} = \partial F_{i,j,k}/\partial u_{r,s,t}$ and observe, as in the 2D case, that $J_{i,j,k,r,s,t}$ is very sparse: $J_{i,j,k,r,s,t} \neq 0$ only for $r = i \pm i$, $s = j \pm 1$, and $t = k \pm 1$ as well as $r = i$, $s = j$, and $t = k$.
Filename: `nonlin_heat_FD_CN_2D`.

Problem 14: Find the sparsity of the Jacobian

Consider a typical nonlinear Laplace term like $\nabla\cdot\alpha(u)\nabla u$ discretized by centered finite differences. Explain why the Jacobian corresponding to this term has the same sparsity pattern as the matrix associated with the corresponding linear term $\alpha\nabla^2 u$.

Hint. Set up the unknowns that enter the difference equation at a point (i, j) in 2D or (i, j, k) in 3D, and identify the nonzero entries of the Jacobian that can arise from such a type of difference equation.
Filename: `nonlin_sparsity_Jacobian`.

Problem 15: Investigate a 1D problem with a continuation method

Flow of a pseudo-plastic power-law fluid between two flat plates can be modeled by

$$\frac{d}{dx}\left(\mu_0\left|\frac{du}{dx}\right|^{n-1}\frac{du}{dx}\right) = -\beta, \quad u'(0) = 0,\ u(H) = 0,$$

where $\beta > 0$ and $\mu_0 > 0$ are constants. A target value of n may be $n = 0.2$.

a) Formulate a Picard iteration method directly for the differential equation problem.

b) Perform a finite difference discretization of the problem in each Picard iteration. Implement a solver that can compute u on a mesh. Verify that the solver gives an exact solution for $n = 1$ on a uniform mesh regardless of the cell size.

c) Given a sequence of decreasing n values, solve the problem for each n using the solution for the previous n as initial guess for the Picard iteration. This is called a continuation method. Experiment with $n = (1, 0.6, 0.2)$ and $n = (1, 0.9, 0.8, \ldots, 0.2)$ and make a table of the number of Picard iterations versus n.

d) Derive a Newton method at the differential equation level and discretize the resulting linear equations in each Newton iteration with the finite difference method.

e) Investigate if Newton's method has better convergence properties than Picard iteration, both in combination with a continuation method.

References

[1] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. SIAM, 1995.

[2] H. P. Langtangen. *Finite Difference Computing with Exponential Decay Models*. Lecture Notes in Computational Science and Engineering. Springer, 2016. `http://hplgit.github.io/decay-book/doc/web/`.

[3] H. P. Langtangen and S. Linge. *Finite Difference Computing with Partial Differential Equations*. 2016. `http://hplgit.github.io/fdm-book/doc/web/`.

Index