Learning Outcomes for Computing Competence

Hans Petter Langtangen[1,2]

Morten Hjorth-Jensen[3,4]

Anders Malthe-Sørenssen[3]

Knut Mørken[5]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo
[3]Department of Physics, University of Oslo
[4]Department of Physics and Astronomy, Michigan State University
[5]Department of Mathematics, University of Oslo

Dec 16, 2015

HTML version
This note is under development! Send comments to `hpl@simula.no`.

Abstract

This note lists a set of learning outcomes for mastering computer-based problem solving in mathematical subjects. A case study describes and discusses the learning outcomes in depth.

Contents

Why is computing competence important?

First of all, we need to define the term *computing* and what it contains.

Definition of computing.

Computing means in this document *solving scientific using computers*. It covers numerical as well as symbolic computing. Computing is also about developing an understanding of the scientific process by enhancing the algorithmic thinking when solving problems.

Computing competence has always been a central part of the science and engineering education. Traditionally, such competence meant mastering mathematical methods to solve science problems - by pen and paper. In 2015, our candidates are expected to use all available tools to solve scientific problems; computers primarily, but also pen and paper. Below, we use the term *algorithms* in the broad meaning: mathematical methods to solve science problems, with and without computers.

Computing competence is about

- derivation, verification, and implementation of algorithms
- understanding what can go wrong with algorithms
- overview of important, known algorithms
- understanding how algorithms are used to solve mathematical problems
- reproducible science and ethics
- algorithmic thinking for gaining deeper insights about scientific problems

Algorithms involving pen and paper are traditionally aimed at what we often refer to as *continuous models*. Application of computers calls for approximate *discrete models*. Much of the development of methods for continuous models are now being replaced by methods for discrete models in science and industry, simply because much larger problem classes can be addressed with discrete models, often also by simpler and more generic methodologies. However, verification of algorithms and understanding their limitations requires much of the classical knowledge about continuous models.

So, why should basic university education undergo a shift from classical mathematics to modern computing?

1. The impact of the computer on mathematics is tremendous: science and industry now rely on solving mathematical problems through computing.
2. Computing increases the relevance in education by solving more realistic problems earlier.
3. Computing through programming is excellent training of creativity.
4. Computing enhances the understanding of abstractions and generalization.

5. Computing decreases the need for special tricks and tedious algebra, and shifts the focus to problem definition, visualization, and "what if" discussions.

The result is a deeper understanding of mathematical modeling. Not only is computing via programming a very powerful tool, it also a great pedagogical aid. We believe in the famous quote by Kristen Nygaard: "Programming is understanding".

For the mathematical training, there is one major new component among the arguments above: *understanding abstractions and generalization*. While many of the classical methods developed for continuous models are specialized for a particular problem or a narrow class of problems, computing-based algorithms are often developed for problems in a generic form and hence applicable to a large problem class.

Key principle in scientific modeling.

The power of the scientific method lies in identifying a given problem as a special case of an abstract class of problems, identifying general solution methods for this class of problems, and applying a general method to the specific problem (applying means, in the case of computing, calculations by pen and paper, symbolic computing, or numerical computing by ready-made and/or self-written software). This generic view on problems and methods is particularly important for understanding how to apply available, generic software to solve a particular problem.

Computing competence represents a central element in scientific problem solving, from basic education and research to essentially almost all advanced problems in modern societies. Computing competence is simply central to further progress. It enlarges the body of tools available to students and scientists beyond classical tools and allows for a more generic handling of problems. Focusing on algorithmic aspects results in deeper insights about scientific problems.

Today's project in science and industry tend to involve larger teams. Tools for reliable collaboration must therefore be mastered (e.g., version control systems, automated computer experiments for reproducibility, software and method documentation).

General learning outcomes for computing competence

Learning outcomes for *numerical algorithms*:

- Deep knowledge of the most fundamental algorithms for linear algebra, ordinary and partial differential equations, optimization, and statistical uncertainty quantification.

- Overview of advanced algorithms and how they can be accessed in available software.
- Knowledge of high-performance computing elements: memory usage, vectorized and parallel algorithms.
- Understanding of approximation errors.
- Application of fundamental and advanced algorithms to classical model problems as well as real-world problems with assessment of the uncertainty in the answer.

Learning outcomes for *symbolic computing*:

- Knowledge of at least one computer algebra system and how it is applied to perform classical mathematics (calculus, linear algebra, differential equations - with verification).

Learning outcomes for *programming*:

- Extensive experience with programming in a high-level language (MATLAB, Python, R). Experience with programming in a compiled language (Fortran, C, C++).
- Extensive experience with implementing and applying numerical algorithms in reusable software that acknowledges the generic nature of the mathematical algorithms.
- Knowledge of basic software engineering elements: functions, classes, modules/libraries, testing procedures and frameworks, scripting for automated and reproducible experiments, documentation tools, and version control systems.
- Extensive experience with debugging software, e.g., as part of implementing comprehensive tests.

Learning outcomes for *verification*:

- Extensive experience with programming of testing procedures.
- Deep knowledge of testing/verification methods:
  - Exact solution of numerical models
  - Method of manufactured solutions (choose solution and fit a problem)
  - Classical analytical solutions (incl. asymptotic solutions)
  - Computing of asymptotic approximation errors (convergence rates)
- Step-wise construction of tests to aid debugging.

Learning outcomes for *mathematical modeling*:

- Experience with deriving computational models from basic principles in applied sciences (physics, geology, biology, etc.).
- Experience with bringing models on dimensionless form to reduce and simplify input data and increase the understanding of the model by interpreting its dimensionless parameters.
- Experience with solving real problems from applied sciences.

Learning outcomes for *presentation of results*:

- Experience with different visualization techniques for different types of computed data.
- Extensive experience with presenting computed results in scientific reports and oral presentations.

What is *deep knowledge*?

By deep knowledge we here mean the understanding of the underlying fundamental ideas and concepts from which a plethora of seemingly different methods and technologies can be derived. In other words, the deep knowledge brings structure to all the technical details.

Obtaining this type knowledge requires time in class and a lot of exercises. In addition, the students need to *reflect* about theory and practice. The reflection process is often difficult to implement. Below are some suggestions.

A useful concept is *simplify, understand, and then generalize.* Giving a superficial overview of a bunch of unrelated methods and their applications to unrelated scientific problems equips the students with a wide toolbox, but fails to enhance a fundamental understanding of how multidisciplinary topics play together. Instead, we believe in the following list.

1. Pick a few selected classes of problems,
2. start out with simplified models,
3. apply general, fundamental ideas to construct algorithms,
4. understand all details to correctly implement the algorithms,
5. understand how to judge the numerical quality of the algorithms,
6. understand how to verify that the computations are mathematically correct.

The verification process forces the student to reflect on all the points: What type of problem is actually solved? How can I test that the solution is right?

After obtaining an understanding of the simplified problem, one can generalize the models to real applications, but illustrate how the insight from the simplified models and methods gives very valuable knowledge when attacking the generalizations. The focus on simplified models help to detach the mathematics from a lot of discipline-dependent application details and cultivate the common mathematical and implementational ideas.

This philosophy is closely related to the *Key principle* stated earlier:

1. solving a complicated problem first starts with the purpose of breaking up the problem into subtasks that belong to general classes of well-studied problems in mathematics,
2. each subproblem is understood with great help simplified models in that class,
3. and finally a synthesis of the subproblems can solve the original problem.

hpl 1: Need to highlight educational methods: instruction based teaching, project work, ...

Case study

The series of goals above are briefly stated, but illustrated here in detail for a special, simple case study: numerical integration by the Trapezoidal rule.

Many science courses now have examples and exercises involving implementation and application of numerical methods. How to structure and verify such numerical programs has, unfortunately, received little attention in university education and the literature. Students and teachers occasionally write programs that are too tailored to the problem at hand instead of being a good starting point for future extensions, and testing is often limited to running a case where the answer seems reasonable. The standards of computing need to be raised to the levels found in experimental physics, chemistry, and biology.

Observation: poor versus good design of programs depends on the programming language (!).

A common conception is that simple scientific computing scripts implemented in Matlab and Python are very similar - almost identical. However, practice observed by this author shows that students and teachers tend to make software with bad design in Matlab, while the design improves significantly when they use Python. Bad design means specializing a generic algorithm to a specific problem and making "flat" programs without functions. Good design means reusable implementations of generic algorithms and proper use of functions (or classes). The coming text demonstrates the assertions.

Exercise for the case study

Integrate the function $g(t) = \exp\left(-t^4\right)$ from -2 to 2 using the Trapezoidal rule, defined by

$$\int_a^b f(x)dx \approx h\left(\frac{1}{2}(f(a)+f(b)) + \sum_{i=1}^{n-1} f(a+ih)\right), \quad h = (b-a)/n \qquad (1)$$

Solution 1: Minimalistic Matlab

Many will attempt to solve the problem by this simple program in Matlab:

```
a = -2; b = 2;
n = 1000;
h = (b-a)/n;
s = 0.5*(exp(-a^4) + exp(-b^4));
for i = 1:n-1
    s = s + exp(-(a+i*h)^4);
end
r = h*s;
r
```

The solution is minimalistic and correct. Nevertheless, this solution has a common pedagogical and software engineering flaw: a special function $\exp(-t^4)$ is merged into a general algorithm (1) for integrating an arbitrary function $f(x)$.

The writer of the program runs it and reports the result: 1.81280494737. How can one assess that this result is correct? There is no exactly known result to compare with. Also, the program above is not well suited for switching to an integrand where we can compare with an exact answer, because several lines need modification.

Solution 2: Matlab with functions

A fundamental software engineering practice is to use *functions* for splitting a program into natural pieces, and if possible, make these functions sufficiently general to be reused in other problems. In the present problem we should strive for the following principles:

1. Since the formula for the Trapezoidal rule works for "any" function, the implementation of the formula should be in terms of a *function* taking $f(x)$, a, b, and n as arguments.

2. The special $g(t)$ formula is implemented as a separate function.

3. A main program solves the *specific problem in question* by calling the *general algorithm* from point 1 with the special data of the given problem ($g(t)$, $a = -2$, $b = 2$, $n = 1000$).

4. Before we can believe in the integration of $g(t)$, we need to *verify* the implementation (see Section).

Let us apply the desirable principles 1-3 in a Matlab context. User-defined Matlab functions must be placed in separate files. This is sometimes found annoying, and therefore many students and teachers tend to avoid functions. In the present case, we should implement the Trapezoidal method in a file `Trapezoidal.m` containing

```
function r = Trapezoidal(f, a, b, n)
% Numerical integration from a to b
% with n intervals by the Trapezoidal rule
f = fcnchk(f);
h = (b-a)/n;
s = 0.5*(f(a) + f(b));
for i = 1:n-1
    s = s + f(a+i*h);
end
r = h*s;
```

The special $g(t)$ function can be implemented in a separate file `g.m` or put in the main program. The function becomes

```
function v = g(t)
v = exp(-t^4)
end
```

Finally, a specialized main program (`main.m`) solves the problem at hand:

```
a = -2; b = 2;
n = 1000;
result = Trapezoidal(@g, a, b, n);
disp(result);
exit
```

The important feature of this solution is that `Trapezoidal.m` can be reused for "any" integral. In particular, it is straightforward to also integrate an integrand where we know the exact result.

An advantage of having the $g(t)$ as a separate function is that we can easily send this function to a different integration method, e.g., Simpson's rule.

Solution 3: Standard Python

Both Solution 1 and Solution 2 are readily implemented in Python. However, functions in Python do *not* need to be located in separate files to be reusable, and therefore there is no psychological barrier to put a piece of code inside a function. The consequence is that a Python programmer is more likely to go for Solution 2. (This may be the reason why the author has observed scientific Python codes to have better design than Matlab codes - modularization comes more natural.) The relevant code can be placed in a single file, say `main.py`, looking as follows:

```
def Trapezoidal(f, a, b, n):
    h = (b-a)/float(n)
    s = 0.5*(f(a) + f(b))
    for i in range(0,n,1):
        s = s + f(a + i*h)
    return h*s

from math import exp  # or from math import *
def g(t):
    return exp(-t**4)

a = -2;  b = 2
n = 1000
result = Trapezoidal(g, a, b, n)
print result
```

This solution acknowledges the fact that the implementation is a generally applicable function, just as the Trapezoidal formula.

However, a small adjustment of this implementation will make it much better. If somebody wants to reuse the `Trapezoidal` function for another integral, they can import `Trapezoidal` from the `main.py` file, but the special problem will be executed as part of the import. This is not desired behavior when solving another problem. Instead, our special exercise problem should be isolated in its own function and called from a test block in the file (to avoid being executed as part of an import). This is the general software design of *modules* in Python.

We therefore rewrite the code in a new file `Trapezoidal.py`:

```
def Trapezoidal(f, a, b, n):
    h = (b-a)/float(n)
    s = 0.5*(f(a) + f(b))
    for i in range(1,n,1):
        s = s + f(a + i*h)
    return h*s

def _my_special_problem():
    from math import exp
    def g(t):
        return exp(-t**4)

    a = -2;  b = 2
    n = 1000
    result = Trapezoidal(g, a, b, n)
    print result
```

```python
if __name__ == '__main__':
    _my_special_problem()
```

Now we have obtained the following important features:

- The file `Trapezoidal.py` is a module offering the widely applicable function `Trapezoidal` for integrating "any" function.
- If `Trapezoidal.py` is run as a program, the if test is true and the special integral of g is computed.
- In an import like `from Trapezoidal import Trapezoidal`, the if test is false and nothing gets computed.

Verification and testing frameworks

An integral part of any implementation is verification, i.e., to bring evidence that the program works correctly from a mathematical point of view. (A related term, validation, refers to bringing evidence that the program produces results in accordance with observations of nature, but this is not of direct interest in this integration context.)

The intuitive approach to testing is to compare results of a program with known mathematical results. For example, we can choose some function, say $\sin t$, and differentiate it to obtain an integrand that we can easily integrate by hand and thereby get a precise number for the integral. Integrating $\int_{-2}^{2} \cos t \, dt$ gives the exact result 1.81859485365. The program with the Trapezoidal rule reports 1.81859242886, so the error $2.42 \cdot 10^{-6}$ is "small". However, we have no idea if this error is just the approximation error in the numerical method or if the program has a bug too! What if the error were $1.67 \cdot 10^{-3}$? It is impossible to say whether this answer is the correct numerical result or not. Actually, this error contains both the approximation error and a bug where the loop goes over $0, 1, \ldots, n-1$.

So, comparison of a numerical approximation with an exact answer does not say much unless the error is "huge" and therefore clearly points to fundamental bugs in the code.

For most numerical methods there are only two good verification methods:

1. Computation of a problem where the approximation error vanishes.
2. Empirical measurement of the convergence rate.

Verification methods should be implemented in *test functions* that can be run at any time to check if the implementation is correct.

A simple test function

The Trapezoidal rule is obviously exact for linear integrands. Therefore, we should test an "arbitrary" linear function and check that the error is close to machine precision. This is done in a separate function in a separate file test_Trapezoidal.py:

```
from Trapezoidal import Trapezoidal
from Trapezoidal_vec import Trapezoidal as Trapezoidal_vec

def linear():
    """Test linear integrand: exact result for any n."""

    def f(x):
        return 8*x + 6

    def F(x):
        """Anti-derivative of f(x)."""
        return 4*x**2 + 6*x

    a = 2
    b = 6
    exact = F(b) - F(a)
    numerical = Trapezoidal(f, a, b, n=4)
    error = exact - numerical
    print '%.16f' % error
```

The output of calling `linear()` is in this case zero exactly, but in general one must expect some small rounding errors in the numerical and exact result.

A proper test function for the nose or pytest test framework

The function `linear` performs the test, but it would be better to integrate the test into a *testing framework* such that we with one command can execute a comprehensive set of tests. This makes it easy to run all tests after every small change of the software. Students should adopt such compulsory habits from the software industry.

The dominating type of test frameworks today is based on what is called *unit testing* in software engineering. It means that we pick a unit in the software and write a function (or class) that runs the test after certain specifications:

- The test function must start with `test_`.
- The test function cannot have any arguments.
- If the test fails, an `AssertionError` exception (in Python) is raised, otherwise the function runs silently.

There are two very popular test frameworks in the Python world now: pytest and nose. There are similar frameworks developed for Matlab too, see a video, but they are not as user friendly since they require the programmer to embed tests in classes (this is still the dominating method in most programming languages).

Using test functions instead of test classes requires writing less code and is easier to learn.

In our case, a proper test function means the following rewrite of the function `linear`:

```
def test_linear():
    """Test linear integrand: exact result for any n."""

    def f(x):
        return 8*x + 6

    def F(x):
        """Anti-derivative of f(x)."""
        return 4*x**2 + 6*x

    a = 2
    b = 6
    expected = F(b) - F(a)
    tol = 1E-14
    computed = Trapezoidal(f, a, b, n=4)
    error = abs(expected - computed)
    msg = 'Trapezoidal: expected=%g, computed=%g, error=%g' % \
          (expected, computed, error)
    assert error < tol, msg
```

The code is basically the same, but we comply to the rules in the three bullet points above. The `assert` statement has the test as `error < tol`, with `msg` as an optional message that is printed only if the test fails (`error < tol` is `False`). The `msg` string can be left out and it suffices to do `assert error < tol`.

The reason why we comply to testing frameworks is that we can use software like nose or pytest to automatically find all our tests and execute them. We put tests in files or directories starting with `test` and run one of the commands

```
Terminal> nosetests -s -v .
Terminal> py.test -s -v .
```

All functions with names `test_*()` in all files `test*.py` in all subdirectories with names `test*` will be run, and statistics about how many tests that failed will be printed. The tests should be run after every modification of the software.

Use of symbolic computing for exact results

We integrated by hand the linear function used in the test above. In more complicated cases it would be safer to use symbolic computing software to carry out the mathematics. Here we demonstrate how to use the Python package SymPy to do the integration:

```
def test_linear_symbolic():
    """Test linear integrand: exact result for any n."""
    import sympy as sym
    # Define a linear expression and integrate it
    x = sym.symbols('x')
    f = 8*x + 6                  # Integrand for this test
    F = sym.integrate(f, x)
```

```python
    # Verify symbolic computation: F'(x) == f(x)
    assert sym.diff(F, x) == f
    # Transform expressions f and F to Python functions of x
    f = sym.lambdify([x], f, modules='numpy')
    F = sym.lambdify([x], F, modules='numpy')

    # Run one test with fixed a, b, n, for scalar and
    # vectorized implementation
    a = 2
    b = 6
    expected = F(b) - F(a)
    tol = 1E-14
    for func in Trapezoidal, Trapezoidal_vec:
        computed = func(f, a, b, n=4)
        error = abs(expected - computed)
        msg = 'expected=%g, computed=%g, error=%g' % \
              (expected, computed, error)
        assert error < tol, msg
```

Note that we now also test both the scalar and the vectorized implementations of the Trapezoidal rule (see Section for explanation of the vectorized version `Trapezoidal_vec`). It is easy in Python to loop over functions (with a variable like `func`). We could also just compare the result of `Trapezoidal_vec` to that of `Trapezoidal` when the latter is verified against the `expected` value.

Use relative errors

Let us change the integration limits in our test example to $a = 2 \cdot 10^8$ and $b = 6 \cdot 10^9$. The computed error in this case is 16384 (!). Hence the tolerance must be set to (e.g.) $2{\cdot}10^5$ (!). In general, the tolerance depends on the magnitude of the numbers involved in the computations. To avoid this dependence, one should use *relative errors*:

```python
error = abs(expected - computed)/abs(expected)
```

Now, a tolerance of 10^{-14} is adequate for the test even if the numbers `expected` and `computed` are large.

A function `test_linear_symbolic_large_limits` in the file `test_Trapezoidal.py` is a test function for a case with large limits and use of the relative error.

Test function for the convergence rate

Let us extend the verification with a case where we know the exact answer of the integral, but we do not know the approximation error. The only knowledge we usually have about the approximation error is of asymptotic type. For example, for the Trapezoidal rule we have an expression for the error from numerical analysis:

$$E = -\frac{(b-a)^3}{12n^2} f''(\xi), \quad \xi \in [a, b].$$

Since we do not know ξ, which is some number in $[a, b]$, we cannot compute E. However, we realize that the error has an asymptotic behavior as n^{-2}:

$$E = Cn^{-2},$$

for some unknown constant C. If we compute two or more errors for different n values, we can check that the error decays as n^{-2}. In general, when verifying the implementation of a numerical method with discretization parameter n, we write $E = Cn^r$, estimate r, and compare with the exact result (here $n = -2$).

More precisely, we perform a set of experiments for $n = n_0, n_1, \ldots, n_m$, where we empirically estimate r from two consecutive experiments:

$$E_i = Cn_i^r,$$
$$E_{i+1} = Cn_{i+1}^r.$$

Dividing the equations and solving with respect to r gives

$$r = \frac{\ln(E_i/E_{i+1})}{\ln(r_i/r_{i+1})}$$

As $i = 0, \ldots, m-1$, the r values should approach the value -2.

It is easy to use the *method of manufactured solutions* to construct a test problem. That is, we first choose the solution, say the integral is given by $F(b) - F(a)$, where

$$F(x) = e^{-x}\sin(2x).$$

Then we fit the problem to accept this solution. In the present case it means that the integrand must be $f(x) = F'(x)$. We use for safety symbolic software to calculate $f(x)$. Thereafter, we run a series of experiments where n is varied, we compute the corresponding convergence rates r from two consecutive experiments and test if the final r, corresponding to the two largest n values, is sufficiently close to the expected convergence rate -2:

```
def test_convergence_rate():
    import sympy as sym
    # Construct test problem
    x = sym.symbols('x')
    F = sym.exp(-x)*sym.sin(2*x)  # Anti-derivative
    f = sym.diff(F, x)            # Integrand for this test
    # Turn to Python functions
    f = sym.lambdify([x], f, modules='numpy')
    F = sym.lambdify([x], F, modules='numpy')

    a = 0.1
    b = 0.9
    expected = F(b) - F(a)
    # Run experiments (double n in each experiment)
    n = 1
    errors = []
    for k in range(28):
        n *= 2
        computed = Trapezoidal(f, a, b, n)
        error = abs(expected - computed)
```

```
        errors.append((n, error))
        print k, n, error
    # Compute empirical convergence rates
    from math import log as ln
    estimator = lambda E1, E2, n1, n2: ln(E1/E2)/ln(float(n1)/n2)
    r = []
    for i in range(len(errors)-1):
        n1, E1 = errors[i]
        n2, E2 = errors[i+1]
        r.append(estimator(E1, E2, n1, n2))
    expected = -2
    computed = r[-1]  # The "most" asymptotic value
    error = abs(expected - computed)
    tol = 1E-3
    msg = 'Convergence rates: %s' % r
    print errors
    assert error < tol, msg
```

The empirical convergence rates are in this example

```
-2.022, -2.0056, -2.0014, -2.00035, -2.000086, -2.000022,
-2.0000054, -2.0000013, -2.00000033
```

Although the rates are known to approach -2 as $n \to \infty$, the rates are close to -2 even for large n (such as $n = 4$). A rough tolerance is often used for convergence rates, for instance 0.1, but here we may use a smaller one if desired.

Summary.

Knowing an exact solution to a mathematical problem and comparing the program output with such a solution, gives only an indication that the program may be correct, but it is only a rough indication. Any real test must use what we know about the approximation error, and that is usually only an asymptotic behavior as function of discretization parameters. The test needs to vary the discretization parameter(s) to estimate convergence rates for comparison with known asymptotic results.

Known analytical solutions are of value in convergence rate tests, but if they are not available, or restricted to very simplified cases, the method of manufactured solutions, where we solve a perturbed problem fitted to a constructed exact solution, is also a very useful technique.

Tests in Matlab

In Matlab, one must decide whether to use a class-based system for unit testing or just write test functions that mimic the behavior of the Python test functions for the nose and pytest frameworks. Here is an example on doing the `test_linear()` function in Matlab:

```
function test_trapezoidal_linear
 %% Check that linear functions are integrated exactly
```

```
  f = @(x) 8*x + 6;
  F = @(x) 4*x**2 + 6*x;  %% Anti-derivative
  a = 2;
  b = 6;
  expected = F(b) - F(a);
  tol = 1E-14;
  computed = trapezoidal(f, a, b, 4);
  error = abs(expected - computed);
  assert(error < tol, 'n=%d, error=%g', n, error);
  end
end

test_trapezoidal_linear()
```

There is, unfortunately, no software available to run all tests in all files in all subdirectories and report on the success/failure statistics, but it is quite straightforward to write such software.

Rounding errors

Verification in terms of measuring convergence rates usually gives a very good insight into approximation errors, but the verification results may be affected by rounding errors, depending on the type of algorithm. For the scalar implementation of the Trapezoidal rule, rounding errors start to affect the results around $n = 2^{24} \approx 16$ million points. Other algorithms are much more sensitive to rounding errors. For example, a numerical derivative like

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x+h)}{h^2},$$

may be subject to rounding errors for moderate values of h. Here is an example with $f(x) = x^{-6}$. An exact answer is $f''(1) = 42$, but numerical experiments for with $h = 10^{-k}$ for various k values end up with

k	numerical f''
1	44.61504
2	42.02521
3	42.00025
4	42.00000
5	41.99999
6	42.00074
7	41.94423
8	47.73959
9	-666.13381
10	0.00000
11	0.00000
12	-666133814.8
13	66613381477.5
14	0.00000

The error starts to increase rather than decrease for $h > 10^{-5}$, and this is because the rounding error is (much) bigger than the approximation error in the formula.

Incorporation of other learning outcomes

We discuss here how some of the learning outcomes from Section can be incorporated in the exercise with the Trapezoidal rule. We restrict programming examples to use Python.

High-performance computing: vectorization

This author has seen a lot of programs used for teaching which apply vectorization without explicit notice. Vectorization is a technique in high-level languages like IDL, MATLAB, Python, and R for removing loops and speed up computations. Unfortunately, the “distance” from the mathematical algorithm to vectorized code is larger than to a plain loop as we used above. Vectorization therefore tends to confuse students who are not well educated in the techniques. For example, the Trapezoidal rule can be vectorized as

```
import numpy as np

def Trapezoidal(f, a, b, n):
    x = np.linspace(a, b, n+1)
    return (b-a)/float(n)*(np.sum(f(x)) - 0.5*(f(a) + f(b)))
```

The code is correct, but it takes some thinking to realize why these lines compute the formula (1). Because of the `sum` function, we need to adjust the summation result such that the weight of the end points becomes correct.

Tip: Implement scalar code first - then vectorize.

It is much easier to get a scalar code, with explicit loops that mimic the mathematical formula(s) as closely as possible, to work first. Then remove loops by vectorized expressions and test the code against the scalar version.

High-performance computing: memory usage

The scalar implementation of the Trapezoidal rule computes one `f(x)` at the time and uses very little memory, actually only 4 `float` variables. The vectorized version, however, computes the function values at all points `x` ($n+1$ `float` objects) at once and therefore requires the storage of about n `float` objects. This is a significant difference between the vectorized and scalar versions. The vectorized version may run out of memory if we want very accurate results and hence a large n.

High-performance computing: parallelization

An important observation for parallelization of the Trapezoidal rule is that all the function evaluations are independent of each other so these can be performed in parallel. Typically, with m compute units we can distribute `int(n/m)` function evaluations to the first $m-1$ units and `int(n/m) + n % m` to the last one. Each unit must compute the sum of the evaluations and communicate to one master unit or to all other units. The master or all units must then sum up all the partial sums, subtract $\frac{1}{2}(f(a)+f(b))$ and multiply by h to get the final answer.

Vectorized algorithms often lend themselves to automatic parallelization. In fact, the Numba tool can automatically parallelize Numerical Python code. Looking at the vectorized Trapezoidal implementation

```
x = np.linspace(a, b, n+1)
I = (b-a)/float(n)*(np.sum(f(x)) - 0.5*(f(a) + f(b)))
```

and assuming that `n` is large, we realize that `np.linspace` must create the vector on m compute units, each with its own memory. In the next expression, `f(x)` leads to application of `f` on the piece of `x` that is on each compute unit. Then `np.sum` creates partial sums of `f(x)` on each compute unit and distributes the results to all other units. No more (distributed) vectors are involved, so the remaining scalar operations can be carried out on every unit, and the final result of the integral is then available on each individual compute unit.

We think it is fundamental that such reasoning is well known among students. Traditionally, thinking about parallelism has not been in focus unless also demanding technical implementations in terms of MPI is also taught. However, laptops will soon be powerful parallel computing platforms, so knowing how to write code that lend itself to easy parallelization by tools such as NumPy and Numba is key. How parallel code is actually implemented may be pushed to a more specialized courses.

Understanding of approximation errors

Since the asymptotic behavior of approximation errors is so fundamental for the most common verification technique (i.e., checking convergence rates), students should be well motivated for diving more into the mathematics behind the various formulas they use in test functions.

`https://www.youtube.com/watch?v=ln1L0xbEM3s`

Overview of advanced algorithms

The Trapezoidal rule is primarily a pedagogical tool for obtaining a good understanding numerical integration and what integration is. For professional use, one will apply more sophisticated algorithms, for instance algorithms that deliver an estimate of the integral with a specified error tolerance.

In the scientific Python eco system we have the `quad` method for sophisticated integration, from the famous QUADPACK Fortran library and made available in the SciPy package. Here we integrate $\int_{-2}^{2}\cos t\,dt$ with a relative error tolerance of 10^{-12}:

```
>>> import scipy.integrate
>>> from math import cos
>>> I, error = scipy.integrate.quad(cos, -2, 2, epsrel=1E-12)
>>> I
1.8185948536513632
>>> error
2.4124935390890847e-14
```

Uncertainty quantification

We want to compute $I = \int_a^2 \cos t\,dt$, but a is an uncertain parameter. Suppose a can be modeled as a normally distributed stochastic variable with mean -2 and standard deviation 0.2. What is the corresponding uncertainty in I? The simplest statistics reflecting the uncertainty of I is the mean and the standard deviation. Monte Carlo simulation is the simplest method for computing the uncertainty.

```
from Trapezoidal_vec import Trapezoidal
import numpy as np
N = 100000        # Monte Carlo samples
a = np.random.normal(loc=-2, scale=0.2, size=N)  # N samples of a
I = np.zeros(N)  # Responses (integrals)
for i in range(N):
    I[i] = Trapezoidal(np.cos, a[i], 2, n=1000)
print 'Integral of cos(t) from t=-2 to t=2:', np.sin(2) - np.sin(-2)
print 'Mean value of uncertain integral:', np.mean(I)
print 'Standard deviation of uncertain integral:', np.std(I)
```

The output becomes

```
Integral of cos(t) from t=-2 to t=2: 1.81859485365
Mean value of uncertain integral: 1.80044133926
Standard deviation of uncertain integral: 0.0856614641125
```

Extended exercise

Compute the following integrals with the Midpoint rule, the Trapezoidal rule, and Simpson's rule:

$$
\begin{aligned}
\int_0^{\pi} \sin x\,dx &= 2,\\
\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-x^2} dx &= 1,\\
\int_0^1 3x^2 dx &= 1,\\
\int_0^{\ln 11} e^x dx &= 10,\\
\int_0^1 \frac{3}{2}\sqrt{x} dx &= 1.
\end{aligned}
$$

For each integral, write out a table of the numerical error for the three methods using a n function evaluations, where n varies as $n = 2^k + 1$, $k = 1, 2, ..., 12$.

A Python solution

In the extended problem, Solution 1 is obviously inferior because we need to apply, e.g., the Trapezoidal rule to five different integrand functions for 12 different n values. Then it only makes sense to implement the rule in a separate function that can be called 60 times.

Similarly, a mathematical function to be integrated is needed in three different rules, so it makes sense to isolate the mathematical formula for the integrand in a function in the language we are using.

We can briefly sketch a compact and smart Python code, in a single file, that solves the extended problem:

```
def f1(x):
    return sin(x)

def f2(x):
    return 1/sqrt(2)*exp(-x**2)

...

def f5(x):
    return 3/2.0*sqrt(x)

def Midpoint(f, a, b, n):
    ...

def Trapezoidal(f, a, b, n):
    ...

def Simpson(f, a, b, n):
    ...

problems = [(f1, 0, pi),  # list of (function, a, b)
            (f2, -5, 5),
            ...
            (f3, 0, 1)]

methods = (Midpoint, Trapezoidal, Simpson)
result = []
for method in methods:
    for func, a, b in problems:
        for k in range(1,13):
            n = 2**k + 1
            I = method(func, a, b, n)
            result.append((I, method.__name__, func.__name__, n))

# write out results, nicely formatted:
for I, method, integrand, n in result:
    print '%-20s, %-3s, n=%5d, I=%g' % (I, method, integrand, n)
```

Note that since everything in Python is an object that can be referred to by a variable, it is easy to make a list `methods` (list of Python functions), and a list `problems` where each element is a list of a function and its two integration limits. A nice feature is that the name of a function can be extracted as a string in the function object (`name` with double leading and trailing underscores).

To summarize, Solution 2 or 3 can readily be used to solve the extended problem, while Solution 1 is not worth much. In courses with many very simple exercises, solutions of type 1 will appear naturally. However, published solutions should employ approach 2 or 3 of the mentioned reasons, just to train students to think that *this is a general mathematical method that I should make reusable through a function*.

Along with the code above there should be a file `test_integration_methods.py` containing test functions for the various rules. The error formula for Simpson's rule contains f'''', so one can integrate a third-degree polynomial in a test and expect an error about the machine precision. The Midpoint rule integrates linear functions exactly. For testing of convergence rates, the Trapezoidal and Midpoint rules have errors behaving as n^{-2}, while the error in Simpson's rule goes like n^{-4}.

Solution 4: a Java OO program

Introductory courses in computer programming, given by a computer science department, often employ the Java language and emphasize object-oriented programming. Many computer scientists argue that it is better to start with Java than Python or (especially) Matlab. But how well is Java suited for introductory numerical programming?

Let us look at our first integration example, now to be solved in Java. Solution 1 is implemented as a simple `main` method in a class, with a code that follows closely the displayed Matlab code. However, students are in a Java course trained in splitting the code between classes and methods. Therefore, Solution 2 should be an obvious choice for a Java programmer. However, it is not possible to have stand-alone functions in Java, functions must be methods belonging to a class. This implies that one cannot transfer a function to another function as an argument. Instead one must apply the principles of object-oriented programming and implement the function argument as a reference to a superclass. To call the "function argument", one calls a method via the superclass reference. The code below provides the details of the implementation:

```
import java.lang.*;

interface Func {  // superclass for functions f(x)
    public double f (double x);  // default empty implementation
}

class f1 implements Func {
    public double f (double t)
    { return Math.exp(-Math.pow(t, 4)); }
}

class Trapezoidal {
    public static double integrate
            (Func f, double a, double b, int n)
```

```
    {
        double h = (b-a)/((double)n);
        double s = 0.5*(f.f(a) + f.f(b));
        int i;
        for (i = 1; i <= n-1; i++) {
            s = s + f.f(a+i*h);
        }
        return h*s;
    }
}

class MainProgram {
    public static void main (String argv[])
    {
        double a = -2;
        double b = 2;
        int n = 1000;
        double result = Trapezoidal.integrate(f, a, b, n);
        System.out.println(result);
    }
}
```

From a computer science point of view, this is a quite advanced solution since it relies on inheritance and true object-oriented programming. From a mathematical point of view, at least when compared to the Matlab and Python versions, the code looks unnecessarily complicated. Many introductory Java courses do not cover inheritance and true object-oriented programming, and without mastering these concepts, the students end up with Solution 1. On this background, one may argue that Java is not very suitable for implementing this type of numerical algorithms.

Conclusions

Simple exercises have pedagogical advantages, but some disadvantages with respect to programming, because the programs may easily become too specialized. In such cases, the exercise may explicitly ask the student to divide the program into functions and make general mathematical methods available as general, reusable functions for a set of problems. This requirement can be motivated by an extended exercise where a piece of code are needed many times, typically that several methods are applied to several problems.

Especially when using Matlab, students may be too lazy to use functions when this is not explicitly required. The result is that testing becomes absent and that extensions to more complicated cases get more error-prone.

Final remark. Linear algebra is an excellent example where the traditional mathematical recipes (algorithms) taught in the standard linear algebra courses all over the world are not compatible with how linear algebra calculations are carried out in the real world. Students are drilled in computing the inverse of 2×2 and 3×3 matrices, computing determinants using co-factors, computing eigenvalues using the characteristic polynomial, and so on. In the real world, however, golden rules of linear algebra go instead as follows:

- Never compute the matrix inverse.
- Never compute a determinant using co-factors.
- Never compute eigenvalues using the characteristic polynomial.
- Never compute $\det A$ to decide if A is approximately singular.
- Never compute the eigenvalues of A to determine whether it is symmetric positive definite. Rather compute the Cholesky factorization; if successful, A is symmetric and positive definite.
- Never compute $A^T A$ when solving least squares problems.

What is the impact of such rules on the teaching of introductory linear algebra?