Documentation for AutoSW Topic 5: Numerical integration methods

Mia Kronner Matr. 22201686

January 27, 2026

Contents

1 Declaration of self reliance

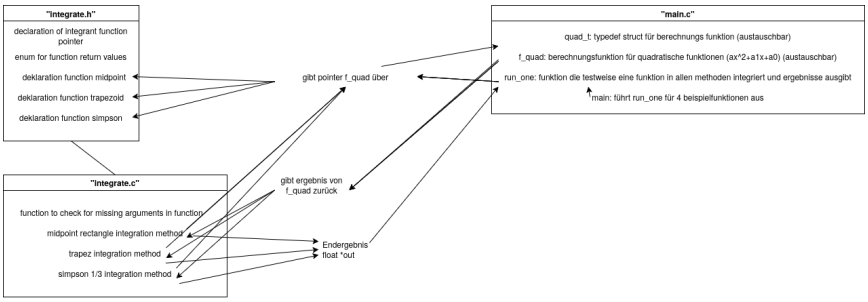
I hereby declare that I have written this thesis independently in accordance with § 35 para. 7 RaPO (Framework Examination Regulations for Universities of Applied Sciences in Bavaria, BayRS 2210-4-1-4-1-WFK), have not submitted it elsewhere for examination purposes, have not used any sources or aids other than those indicated, and have marked all direct and indirect quotations as such.

2 Task desciption

Topic 5: Numerical integration methods The aim of this task is to implement three variants of numerical approaches for calculating integrals in a software module. A function for the rectangle method, the trapezoid method, and the Simpson method should be designed and coded in a reusable module. When defining the interface of the C functions, pay attention to reusability. The task also includes developing a sample application to test the three variants. Select two examples of mathematical functions whose integrals are to be determined, with corresponding integration boundaries, and compare the three methods in terms of their precision. Test the limitations of the methods. Substantiate your results. Explain the mathematical relationships underlying the methods in your paper. A graphical representation will help to illustrate the understanding of the relationships. Optimize the source code as far as possible for performance-optimized use on an embedded controller. The result of the task is a project that can be compiled with MS Studio, which enables the software module to be tested (suitable test cases, etc.) using a command line application, plus the documentation specified below. Scope of relevant task-specific documentation

- Graphical representation of the software design and compact description of the key relationships (1 page)
- Theoretical explanation of the relationships (mathematical, algorithmic, etc.) that need to be taken into account here (2 pages)
- Documentation of the reference examples used for testing, with graphical representation and explanation of method limitations (2 pages)

3 Graphical representation of the software design



4 Theoretical explanation of the relationships

4.1 "integrate.h":

Interface for `integrate.c` Module. Here is declared:

- a void float function pointer for the mathematical function which is to be integrated `integrand_f`.
- a typedef which contains the possible return values for the three integration functions `integ_status_t`
- the three integration functions `midpoint` (rectangle midpoint method), `trapezoid` (trapezoid method), and `simpson` (simpson 1/3 method)
  - they each take a integrand function `integrand_f f`, its context as a void pointer `void *ctx`, the limits `a` and `b` and the number of intevals `n`

```
#ifndef INTEGRATE_H_
#define INTEGRATE_H_

#include <stddef.h>

typedef float (*integrand_f)(float x, void *ctx);

typedef enum
{
    INTEG_OK = 0,
    INTEG_ERR_BAD_ARGS = -1,
    INTEG_ERR_N_EVEN_REQUIRED = -2
} integ_status_t;

integ_status_t midpoint(integrand_f f, void *ctx, float a, float b, unsigned n,
                        float *out);

integ_status_t trapezoid(integrand_f f, void *ctx, float a, float b, unsigned n,
                         float *out);

integ_status_t simpson(integrand_f f, void *ctx, float a, float b, unsigned n,
                       float *out);

#endif // INTEGRATE_H_
```

4.2 "integrate.c":

Implements the three numerical integration algorithms (midpoint/rectangle, trapezoid, Simpson 1/3) behind the public interface from `integrate.h`, including argument validation and method-specific constraints.

The file includes `integrate.h` and provides the concrete implementations of `midpoint()`, `trapezoid()`, and `simpson()`, each computing an approximation of

$$I = \int_a^b f(x)dx$$

by sampling the integrand at specific points and summing weighted contributions.

All three functions follow the same reusable calling convention: function pointer `integrand_f f`, a generic context pointer `ctx`, integration bounds `a`, `b`, subinterval count `n`, and output pointer `out`.

```
static inline int bad_args(integrand_f f, float *a, float *b, unsigned *n,
                           float *out)
{
    return (f == NULL || out == NULL || *n == 0u || !(*b > *a));
}

integ_status_t midpoint(integrand_f f, void *ctx, float a, float b, unsigned n,
                        float *out)
{
    /*check if all necessary parameters have been given */
    if (bad_args(f, &a, &b, &n, out))
        return INTEG_ERR_BAD_ARGS;

    float sum = 0.0f;
    float fx = 0.0f;

    /*intervall width h: */
    const float h = (b - a) / (float)n;
    float x = a + 0.5f * h;

    for (unsigned i = 0; i < n; ++i)
    {
        fx = f(x, ctx);
```

```
        sum = sum + fx;
        x = x + h;
    }

    *out = sum * h;
    return INTEG_OK;
}

integ_status_t trapezoid(integrand_f f, void *ctx, float a, float b, unsigned n,
                         float *out)
{
    if (bad_args(f, &a, &b, &n, out))
    {
        return INTEG_ERR_BAD_ARGS;
    }

    const float h = (b - a) / (float)n;
    float sum = 0.5f * (f(a, ctx) + f(b, ctx));

    float x = a + h;
    for (unsigned i = 1; i < n; ++i)
    {
        sum = sum + f(x, ctx);
        x = x + h;
    }

    *out = sum * h;
    return INTEG_OK;
}

integ_status_t simpson(integrand_f f, void *ctx, float a, float b, unsigned n,
                       float *out)
{
    if (bad_args(f, &a, &b, &n, out))
    {
        return INTEG_ERR_BAD_ARGS;
    }
    if (n & 1u) /*n must be even*/
    {
        return INTEG_ERR_N_EVEN_REQUIRED;
```

```
    }

    float sum_odd = 0.0f;
    float sum_even = 0.0f;
    const float h = (b - a) / (float)n;

    float x = a + h;
    for (unsigned i = 1; i < n; ++i)
    {
        const float fx = f(x, ctx);
        if (i & 1u)
            sum_odd = sum_odd + fx;
        else
            sum_even = sum_even + fx;
        x = x + h;
    }

    *out =
        (h / 3.0f) * (f(a, ctx) + 4.0f * sum_odd + 2.0f * sum_even + f(b, ctx));
    return INTEG_OK;
}
```

4.2.1 Common argument checking (`bad_args`)

A shared internal helper `bad_args()` is implemented as `static inline` to reduce call overhead and to allow the compiler to inline it for performance on embedded targets.

It returns an error condition if `f == NULL, out == NULL, n == 0` or the interval is invalid (the code checks `b > a`). If `bad_args()` reports invalid inputs, each public integration function returns `INTEG_ERR_BAD_ARGS` immediately.

4.2.2 Midpoint / rectangle method (`midpoint`)

1. Mathematical idea

   The interval [a,b] is split into n subintervals of equal width

   $$h = (b - a)/n$$

   , and each subinterval [xi,xi+1] is approximated by a rectangle whose

height is the function value at the midpoint

$$xi + h/2$$

. The Formula is as follows: [1]

$$\approx h \sum_{i=1}^{n} f(x_{i-1} + x_i/2)$$

2. Implementation details The code computes `h`, initializes the first midpoint `x = a + 0.5fh`, then loops exactly `n` times, accumulating `sum = sum + f(x, ctx)` and stepping `x = x + h`.

   Finally, the result is written as `out = sum * h` and the function returns `INTEG_OK`.

4.2.3 Trapezoid method (`trapezoid`)

1. Mathematical idea

   Each subinterval is approximated by a trapezoid formed by the points

   $$(xi, f(xi))$$

   and

   $$(xi + 1, f(xi + 1))$$

   , giving the composite trapezoid rule.

   This corresponds to the standard weighted sum [2]

   $$= h/2(f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n))$$

2. Implementation details

   The code computes `h`, then initializes `sum = 0.5f*(f(a,ctx)+f(b,ctx))` to apply the half-weight to the endpoints. It iterates from the first interior node `x = a + h` for `i = 1..n-1`, adds each interior sample once (`sum = sum + f(x,ctx)`), then scales by `h` via `*out = sum * h`.

4.2.4 Simpson 1/3 method (`simpson`)

1. Mathematical idea

   Simpson's 1/3 rule approximates the function by piecewise quadratic polynomials over pairs of subintervals, which leads to alternating weights 4 and 2 for interior points.

   The composite Simpson rule requires an even number of subintervals nn so that the domain can be grouped into

   $$n/2$$

   pairs. The formula is as such: [3]

   $$\approx (h/3)[f(x_0) + 4f(x_1) + 2f(x_2) + \ldots 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

2. Implementation details and constraint handling

   After basic argument validation, the function checks `if (n & 1u)` and returns `INTEG_ERR_N_EVEN_REQUIRED` if `n` is odd, enforcing the "even n" requirement from the interface contract.

   It then loops over the interior nodes `i=1..n-1` and accumulates two partial sums: `sum_odd` for odd indices (weight 4) and `sum_even` for even indices (weight 2). The final formula implemented is `out = (h/3)(f(a)+4*sum_odd+2*sum_even+f(b))`, which matches the documented Simpson weighting scheme.

4.3 Performance-oriented aspects (embedded focus)

The design avoids dynamic allocation and uses a caller-provided output pointer (`float *out`), which is predictable and typical for embedded C modules.

The `ctx` pointer allows passing coefficients/parameters without global variables, enabling reuse for many function types while keeping the integrator code generic. Using `static inline` for `bad_args` and keeping loop bodies simple (incrementing `x` by `h` rather than recomputing from scratch) supports compiler optimization and reduces runtime overhead.

When checking for evenness of `n` the simpson function uses `& u1` instead of `% 2`. It is not possible to substitue a `/2` for a `>>1` for the float variable type. Such tricks can not have been used for any values with the float type.

5 Documentation of the reference examples used for testing

`"main.c"` is a small command-line test application that demonstrates how to use the reusable integration module by defining example integrand functions, calling all three numerical methods, and printing absolute errors against known exact results.

The application includes `integrate.h` and uses standard library functionality (printing and absolute error via `fabsf`) to compare the numeric results to a known reference ("exact") value. Its purpose is not to be a generic framework, but a compact test harness that exercises the module API and makes method limitations visible at runtime (e.g., Simpson's even-`n` requirement).

5.1 Test integrand design

Parameterized quadratic via context pointer

A small struct `quad_t` stores coefficients `a2`, `a1`, `a0` for a quadratic polynomial

$$f(x) = a_2x^2 + a_1x + a_0$$

, showing how the integrator's `void *ctx` can carry user-defined parameters without globals. The function `f_quad(float x, void ctx)` casts `ctx` to `const quad_t` and evaluates the polynomial, matching the generic `integrand_f` interface expected by the integration module.

5.2 Test runner (`run_one`)

The helper `run_one(...)` calls `midpoint`, `trapezoid`, and `simpson` with the same function, bounds, and number of subintervals, storing results into three local floats (`r`, `t`, `s`). It prints the integration problem setup (function name, bounds, `n`), prints the exact reference value, and prints each method's absolute error using `fabsf(result - exact)` for a direct precision comparison. Simpson's status return is checked: if `simpson` returns `INTEG_OK`, the Simpson result/error is printed, otherwise a message is printed indicating it was not computed because `n` must be even.

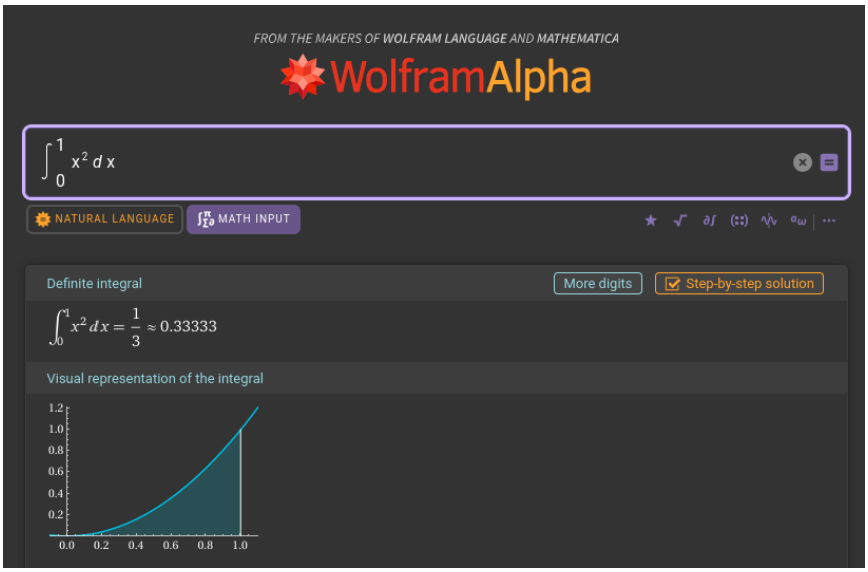
5.3 Reference examples in `main()`

Four concrete test cases are instantiated using `quad_t` coefficients and passed to `run_one`. They have been chosen in such a way that many edge cases are accounted for.

Example 1: `q`

$$f(x) = x^2$$

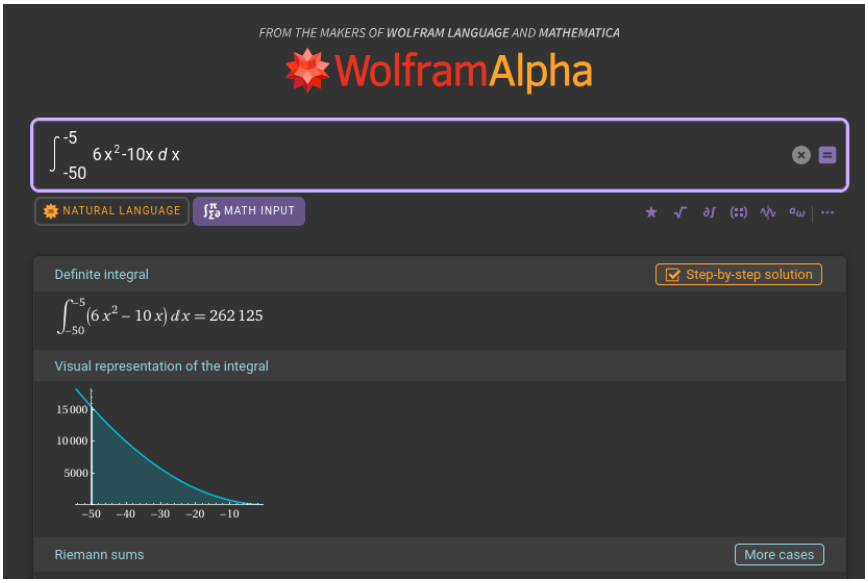
on [0,1] with exact integral $1/3$ and `n=10`



Example 2: `q2`

$$f(x) = -x2 + 3x + 50$$

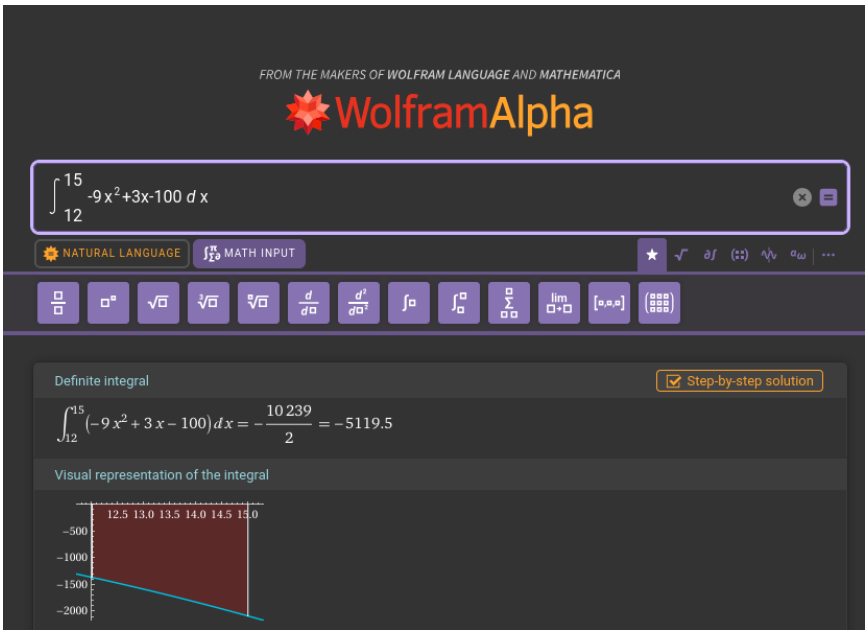
on [-3,15] with exact integral 90 and `n=22`



Example 3: `q3`

$$f(x) = -9x^2 + 3x - 100$$

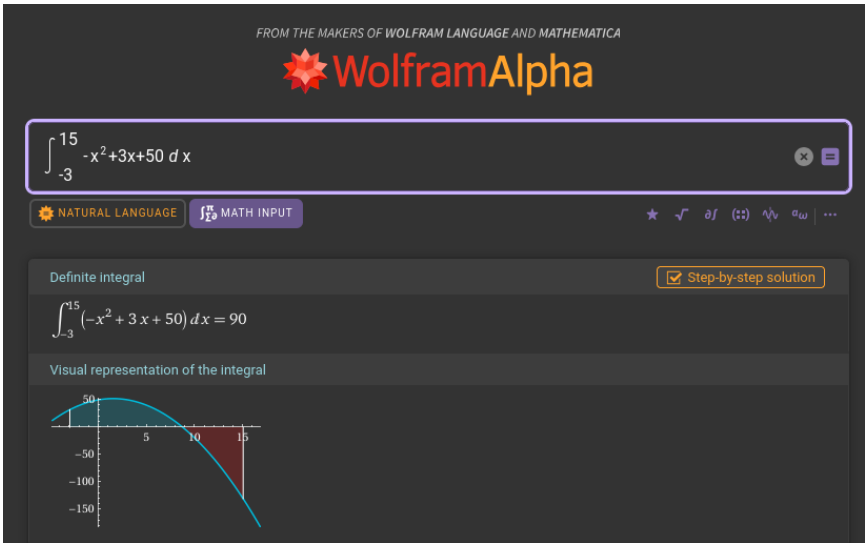
on [12,15] with exact integral -5119.5 and `n=2`



Example 4: `q4`

$$f(x) = 6x^2 - 10x$$

on [-50,-5] with exact integral 262125 and `n=6u`




```
#ifndef INTEGRATE_H_
#define INTEGRATE_H_

#include <stddef.h>

typedef float (*integrand_f)(float x, void *ctx);

typedef enum
{
    INTEG_OK = 0,
    INTEG_ERR_BAD_ARGS = -1,
    INTEG_ERR_N_EVEN_REQUIRED = -2
} integ_status_t;

integ_status_t midpoint(integrand_f f, void *ctx, float a, float b, unsigned n,
                        float *out);

integ_status_t trapezoid(integrand_f f, void *ctx, float a, float b, unsigned n,
                         float *out);

integ_status_t simpson(integrand_f f, void *ctx, float a, float b, unsigned n,
                       float *out);
```

```
#endif // INTEGRATE_H_

static inline int bad_args(integrand_f f, float *a, float *b, unsigned *n,
                           float *out)
{
    return (f == NULL || out == NULL || *n == 0u || !(*b > *a));
}

integ_status_t midpoint(integrand_f f, void *ctx, float a, float b, unsigned n,
                        float *out)
{
    /*check if all necessary parameters have been given */
    if (bad_args(f, &a, &b, &n, out))
        return INTEG_ERR_BAD_ARGS;

    float sum = 0.0f;
    float fx = 0.0f;

    /*intervall width h: */
    const float h = (b - a) / (float)n;
    float x = a + 0.5f * h;

    for (unsigned i = 0; i < n; ++i)
    {
        fx = f(x, ctx);
        sum = sum + fx;
        x = x + h;
    }

    *out = sum * h;
    return INTEG_OK;
}

integ_status_t trapezoid(integrand_f f, void *ctx, float a, float b, unsigned n,
                         float *out)
{
    if (bad_args(f, &a, &b, &n, out))
    {
        return INTEG_ERR_BAD_ARGS;
```

```
    }

    const float h = (b - a) / (float)n;
    float sum = 0.5f * (f(a, ctx) + f(b, ctx));

    float x = a + h;
    for (unsigned i = 1; i < n; ++i)
    {
        sum = sum + f(x, ctx);
        x = x + h;
    }

    *out = sum * h;
    return INTEG_OK;
}

integ_status_t simpson(integrand_f f, void *ctx, float a, float b, unsigned n,
                       float *out)
{
    if (bad_args(f, &a, &b, &n, out))
    {
        return INTEG_ERR_BAD_ARGS;
    }
    if (n & 1u) /*n must be even*/
    {
        return INTEG_ERR_N_EVEN_REQUIRED;
    }

    float sum_odd = 0.0f;
    float sum_even = 0.0f;
    const float h = (b - a) / (float)n;

    float x = a + h;
    for (unsigned i = 1; i < n; ++i)
    {
        const float fx = f(x, ctx);
        if (i & 1u)
            sum_odd = sum_odd + fx;
        else
            sum_even = sum_even + fx;
```

```
        x = x + h;
    }

    *out =
        (h / 3.0f) * (f(a, ctx) + 4.0f * sum_odd + 2.0f * sum_even + f(b, ctx));
    return INTEG_OK;
}

#include <math.h>
#include <stdio.h>

typedef struct
{
    float a2, a1, a0;
} quad_t;

static float f_quad(float x, void *ctx)
{
    const quad_t *q = (const quad_t *)ctx;
    return (q->a2 * x * x) + (q->a1 * x) + q->a0;
}

static void run_one(const char *name, integrand_f f, void *ctx, float a,
                    float b, float exact, unsigned n)
{
    float r = 0.0f, t = 0.0f, s = 0.0f;
    integ_status_t stS;

    midpoint(f, ctx, a, b, n, &r);
    trapezoid(f, ctx, a, b, n, &t);

    stS = simpson(f, ctx, a, b, n, &s);

    printf("\n%s on [%.6f, %.6f], n=%u\n", name, a, b, n);
    printf("Exact:      %.9f\n", exact);
    printf("Midpoint:   %.9f  err=%.9f\n", r, fabsf(r - exact));
    printf("Trapezoid:  %.9f  err=%.9f\n", t, fabsf(t - exact));
    if (stS == INTEG_OK)
        printf("Simpson:    %.9f  err=%.9f\n", s, fabsf(s - exact));
    else
```

```
        printf("Simpson:    not computed (n must be even)\n");
}

int main(void)
{
    const quad_t q = {1.0f, 0.0f, 0.0f}; /*x^2*/

    const quad_t q2 = {-1.0f, 3.0f, 50.0f}; /*x^2+3x+50*/

    const quad_t q3 = {-9.0f, 3.0f, -100.0f}; /*-9x^2+3x-100*/

    const quad_t q4 = {6.0f, -10.0f, 0.0f}; /*6x^2-10x*/

    run_one("x^2", f_quad, (void *)&q, 0.0f, 1.0f, 1.0f / 3.0f, 10u);

    run_one("x^2+3x+50", f_quad, (void *)&q2, -3.0f, 15.0f, 90.0f, 22u);

    run_one("-9x^2+3x-100", f_quad, (void *)&q3, 12.0f, 15.0f, -5119.5f, 2u);

    run_one("6x^2-10x", f_quad, (void *)&q4, -50.0f, -5.0f, 262125.0f, 6u);

    return 0;
}
```

5.3.1 Results

6 Bibliography

[1] "Engineering at Alberta Courses Rectangle Method," Jan. 2026. Available: https://engcourses-uofa.ca/books/numericalanalysis/numerical-integration/rectangle-method

[2] "Engineering at Alberta Courses Trapezoidal Rule," Jan. 2026. Available: https://engcourses-uofa.ca/books/numericalanalysis/numerical-integration/trapezoidal-rule

[3] "Simpson's Rule (Simpson's 1/3 Rule) - Formula, Derivation, Examples," Jan. 2026. Available: https://www.cuemath.com/simpsons-rule-formula