Belegarbeit Labor 4 - Grundlagen der Programmierung HTW

Lennard Wittenberg

18. Januar 2024

Zusammenfassung

Belegarbeit zum Laborblatt 4. Es enthält eine Beschreibung der Aufagbenstellung, Grundenlagen, Vorgehen und Lösungen zu den Laboraufgaben und ein Fazit.

Inhaltsverzeichnis

1 Einführung

Im Rahmen des HTW Moduls CE22 ”Grundlagen der Programmierung Labor-4”beschäftige ich mich in dieser Belegarbeit mit dem Testen und Dokumentieren von Programmen anhand von graphischer Darstellung der Resultate. Zusätzlich werde ich weitere Nutzungsmöglichkeiten des Build-Systems ”make”thematisieren, die die Wierderverwendbarkeit und Nutzerfreundlichkeit deutlich erhöhen. Darüberhinaus gehe ich auch auf die Verwendung von Funktionszeigern ein. Die konkreten Aufgabenstellungen, erforderlichen Grundlagen, mein Vorgehen bei den Problemstellungen sowie die letztendlichen Lösungen werden in den folgenden Abschnitten detailliert beschrieben. Laborblatt 4

1.1 Aufgabenstellung

Das 3.Labor beinhaltet 5 Aufagben.
1.) In der ersten Aufgaben sollen grundlegende Fragen zu den Themen des Labors beantwortet werden. Dies dient zum einen zur Wiederholung einiger vorgegangen Grundlagen, als auch zur Aneignung der neuen Grundlagen.
2.) Makefile vereinfachen. In dieser Aufgabe wird das verbesserte Makefile Konstrukt gebaut. Allerdings wird es noch nicht konkret verwendet.
3.) GCD. Das Projekt gcd –¿ GröSSter Gemeinsamer Teiler wird in dieser Aufgabe um zwei Optionen erweitert sowie mit dem neuen Makefile Konstrukt verbunden.
4.) Numerische Integration. In dieser Aufgabe wird ein Projekt erstellt welches das Integral von mathematischen Funktionen mit einer Variable zu berechnen. 5.) SDL2. In dieser Aufgabe wird das System SDL2 anhand eines kleinen Programms vorgestellt.

2 Grundlagen

In diesem Abschnitt finden Sie alle Informationen und Grundlagen die für die Bearbeitung des 4. Labors nötig sind.

2.1 Variablen in Makefile definieren

Um eine Varibale in einem Makefile zu definieren schreibt man erst den gewünschten Variablen Namen und verbindet diesen mit einem der drei Zeichen = , := , oder ::= mit einem Wert. Meistens werden Variablen verwendet, um Befehlsketten zu vereinfachen. Der Name einer Variable darf die folgenden Zeichen nicht enthalten.
:, #, = oder Leerzeichen
Hier ein kurzes Beispiel:

```
MAINSRC = gcd.c
```

2.2 automatische Variablen in make

Automatische Variablen sind spezifische Variablen, dessen Werte automatisch bassierent auf den jeweiligen Abhänigkeiten und dem Target selbest bestimmt werden.
Beispielsweise $@, $<und $^
sind automatische Variablen.
$@ Diese Variable beinhaltet den Namen des Target einer Reglen.
$<Diese Variable beinhaltet den Namen der ersten Abhängigkeit der Regel.
$^
Diese Variable beinhaltet die Namen aller Abhängigkeiten der Regel mit Leerzeichen dazwischen.

2.3 Substitution References in einem Makefile

Eine Substitution Reference ersätzt den Wert einer Variable mit einer Alternative, die spezifiziert werden kann.
Zum Beispiel:

```
OTHEROBJS = $(OTHERSRCS:.c=.o)
```

OTHEROBJS speichert jetzt die Namen bzw. die Adressen der Objektdateien von allen Programmen, die in OTHEROBJS gespeichert waren.

2.4 addprefix

```
$(addprefix prefix/ names ...)
```

das Argument names kann eine Serie von Namen oder ein einzelner Name sein. Das Argument prefix wird als Einheit verwendet. Der Wert von prefix wird vor jeden individuelen Namen geschrieben. Zwischen dem Prefix mögliches Beispiel:
$(addprefix src/,foo bar)
ergibt src/foo src/bar

2.5 Pattern-Rules(Musterregel)

Eine Musterregel sieht aus wie eine gewöhnliche Regel, außer dass ihr Ziel das Zeichen % (genau eines davon) enthält. Das Ziel wird als Muster für übereinstimmende Dateinamen betrachtet. So kann man anzeigen, wie sich ihre Namen auf den Zielnamen beziehen.
Daher gibt eine Pattern-Rule Für eine genauere Erklärung siehe 5

2.6 Order-only Dependencies

eine Voraussetzung vor einem Ziel erstellt wird, ohne jedoch eine Aktualisierung des Ziels zu erzwingen. Alle Voraussetzungen (prerequisite) links vom Pipe-Symbol (|) sind

normal; Alle Voraussetzungen auf der rechten Seite sind Orderonly:
targets : normal-prerequisites | order-only-prerequisites

2.7 -include Direktive

Die -include Direktive erfüllt die gleiche Funktion wie include. Der Unterschied ist, dass die -include Diektive keine Fehler oder Wahrnungen an das System gibt, wenn bei einem Aufruf eine Datei oder Abhänigkeit einer Datei nicht gefunden, wiederhergestellt werden kann oder nicht exisistiert.

2.8 Compiler Optionen

-MP weist CPP an für jede Abhänigkeit eines Targets auSSer der Hauptdatei ein eigenes .PHONY Target zu erstellen. Dadurch werden die einzelnen Abhänigkeiten unabhänigig voneinander und dem Target selbest. Dadurch kann man Fehler umgehen die Make zurückgibt, für den Fall das Header-Dateien entfernt wurden ohne, dass die Makefile vorher aktualisiert wurde.
-MMD funktioniert genau wie -MD allerdings werden bei -MMD nur von dem Nutzer erstellte Header-Dateien akzeptiert und System-Header-Dateien werden nihct akzeptiert bzw. ignoriert.
-MD ist ein Äquivalent von den Optionen -M und -MF mit dem Unterschied, dass -E nicht impleziert wird. Wird diese Option in Verbindung mit einer Datei aufgerufen überprüft der Treiber, ob in Verbindung mit dieser Datei eine -o Option vorhanden ist. Ist dies der Fall verwendet der Treiber die Datei Argument nur mit dem Suffix .d. Ist dies nicht der Fall entfernt der Treiber alle Verzeichnis Komponenten und Suffixe der eingebenen Datei und fügt den Suffix .d hinzu.
Da -E nicht impliziert ist, kann -MD als Nebeneffekt des Kompilierungsprozesses zum Generieren einer Abhängigkeitsausgabedatei verwendet werden.
-O0: Deaktiviert jegliche Optimierungen. Das ist die Standardeinstellung wenn nichts anderes angegeben wird.
-O1: Optimiert schon etwas mehr als -O und führt auch Optimierungen durch die die etwas länger dauern können.
-O2: Hier werden schon fast alle möglichen Optimierungen bis auf Functioninlining und Loopunrollung durchgeführt.
-O3: Führt alle möglichen Optimierungen wie z.B. auch Functioninlining und Loopunrolling durch.
-Os: Optimiert den Code auf minimale GröSSe. Dabei kann die ausführbare Datei etwas langsamer werden, da zum Beispiel meist kein Funktionsinlining durchgeführt wird, sondern ein Funktionsaufruf gemacht wird, der natürlich langsamer ist.
-Ofast: Optimiert den Code sodass dieser schnller laufen kann.

2.9 address-sanatizer

Ein Code-Sanitizer ist ein Programmiertool, das Fehler in Form von undefiniertem oder verdächtigem Verhalten durch einen Compiler erkennt, der zur Laufzeit Instrumentierungscode einfügt. Um einen solchen Sanitizer zu nutzen, kann man an den den Compiler folgende Liniker-flag schicken:
-fsanitize=address

2.10 Array Repräsentation

eine Matrix bzw. Array werden in der Standardform coloumn-major-layout repräsentiert. →Array[Spalten][Zeilen]. Beim Row-major-layout werden zuerst die Zeilen äufgefüllt", bevor in einer neuen Spalten die nächste Zeile erstellt wird. →Array[Zeilen][Spalten]

2.11 Funktionszeiger

In C/C++ können Funktionen über Zeiger, die die Aufruf-Adresse der Funktion enthalten, aufgerufen werden. Dadurch kann ein Zeiger, wie eine Funktion genutzt werden. Ein Funktionszeiger wird folgendermaSSen deklariert:
typ (*f)(Paramenter-Typ-Liste)

2.12 Strukturen

Eine Struktur struct ist ein abgeleiteter Datentyp, der Elemente unterschiedlichen Typs zusammenfasst. In C werden Strukturen deklariert mit einem Namen (identifier) und einem Array mit Elementen der Struktur. Dabei sind die Elemente jeweils ein eigenes Array mit beliebigen Datentypen.

3 Vorgehensweise

In den folgenden Textabschnitten werden wie in 1.1 beschrieben meine Vorgehensweise bei der Bearbeitung der Laboraufgaben dargestellt.

3.1 Fragen und Grundlagen

Den Großteil meiner Recherce Ergebnisse finden Sie im Teilabschnitt Grundenlagen. Die vollständige Beantwortung aller Fragen und Grundlagen finden Sie in diesem Dokument. Eine Auflistung meiner Quellen finden Sie im Literaturverzeichnis 5.

3.2 Vereinfachung des Makefiles

Zur Bearbeitung dieser Aufgabe gab es eine Anleitung, in der Schritt für Schritt erklärt wird, wie das Makefile verändert und verbessert werden soll. Darüber hinaus wurde ein bereits fast fertiges Makefile zur Verfügung gestellt. Man musste größtenteils nur ein paar

Variablen Namen an das jeweilige Projekt anpassen. Die einizigen Dinge, die konkret selber implemtiert werden mussten, waren zwei Targets. Das eine Target soll die Tests des Projekts ausführen um die Funktionen auf Fehler zu überprüfen. Mit dem Zweitem Target sollte Doxygen durchgeführt werden. Für die Implementation von Doxygen habe ich mich für meine Implemtation aus dem letzten Labor entschieden. Meiner Meinung nach bietet diese Implementation bereits eine sehr gute Wiederverwendbarkeit, da man den Code einfach kopieren und nur den Namen des Ziel Dokuments anpassen muss. Das einzige Problem war die Implementation von test und coverage. Für beide Tragets muss eine For-Schleife geschrieben, welche alle Testobjekte und Funktionsobjekte aufruft. Die Syntax für diese For-Schleifen ist komplizierter als For-Schleifen in C. Lösung:

```
.PHONY: test
test: $(TESTASANTARGETS)
        $(foreach T, $(TESTASANTARGETS), $(T) && ) true

.PHONY: coverage
coverage: $(TESTCOVERTARGETS)
        $(foreach T, $(TESTCOVERTARGETS), $(T) &&) true
        gcovr
```

3.3 GröSSter Gemeinsamer Teiler Erweiterung

3.3.1 Makefile und Struktur

gcd Repository Ziel: Verbesserung des Makefile mit der neuen Version und Programmargumente über eine Strucktur laufen lassen.
Zunächst habe ich das Makefile des Projekts durch die neue Vorlage für Makefiles ersetzt. Hier mussten nur einige Variablen Namen geändert werden. AnschlieSSend kann das Makefile einfach in den Projket Ordner kopiert werden. Bei der Implementation der Struktur cmdargs und der dazu gehörigen Funktion cmdargs_parse() habe ich mich streng an das vorgegebene Beispiel gehalten und ausschließlich neue Variablen und Fallunterscheidungen hinzu gefügt.

3.3.2 Iteration-map

Ziel: Veränderung der Implementation des Euklidischen Algorithmus, sodass die Anzahl der Implementationen zurückgegeben wird und die Darstellung des Alogorithmus in einem Diagramm.
Ich habe anstelle die Implementation des Algorithmus zu verändern eine neue Funktion erstellt, die diese Aufagbe erfüllen soll. Dafür habe ich meine Funktion für die Option –verbose benutzt. Diese habe in so verändert, dass bei jeder Iteration ein Counter inkrementiert wird. Wenn die Iteration abgeschlossen ist, wird der Counter als Rückgabewert zurückgegeben.
Die Implementation der Iterationkarte war für mich ein großes Problem, welches dafür

gesorgt hat, dass sich alles andere stark verzögert hat. die Problemstellen waren die korekte Erstellung des Puffers. Die Lösung war die Funktion malloc(), welche die benötigte Größe für den Puffer errechnet. Darüber hinaus ist wichtig zu erwähnen, dass der Puffer ein eindimensionales row-mayor-layout array ist. Das heißt, dass die Werte für die Iterationen mithilfe einer doppelten for-Schleife berechnet werden. →Es werden alle Iterationen gezählt, für y = 0 und 0 <x <maximaler x-Wert = Programm Argument b. Danach wird y um 1 inkrementiert und der Vorgang wird wiederholt, bis x und y maximal sind. Dieses Ergebnis wird dann mit der Funktion write_pgm() in ein Diagram umgewandelt. Die genaue Funktionsweise von write_pgm ist im Repositoty zu finden.

```
if(gcdargs.iteration_map){
        int n = 0;
        int maxvalue = gcdcount(gcdargs.a,gcdargs.b);
        unsigned char *buf = malloc(gcdargs.a*gcdargs.b);
                for(int i = 1; i < gcdargs.a;i++){
                        for(int j = 1; j < gcdargs.b;j++){
                                buf[n] = gcdcount(i,j);
                                n++;
                        }
                }
write_pgm(gcdargs.iteration_map,buf,100,100,maxvalue);
```

3.3.3 Zeitmessung(Measure-time)

Ziel: Einfügen einer Option, welche die Zeit misst, die benötigt wird, um die Iterationswerte für Iteration-Map zu erstellen.
Dafür wird die Funktion clock-gettime benötigt. Diese kann mit der Header-Datei time.h in das Programm integriert werden. Die benötigten Variablen werden ebenfalls mit der Header-Datei zur Verfügung gestellt. Der zu messende Bereich wird mit zwei clock_gettime() Aufrufen umschlossen, ein Aufruf vor dem Bereich und einer nach dem Bereich. Die gemessenen Werte werden in einer Struktur timespec (ebenfalls aus time.h) gespeichert und können dann auf der Konsole gedrückt werden. Für weitere Informationen über time.h sind in den man Pages man clock_gettime und man timespec zu finden.

```
struct timespec start, finish;
                clock_gettime(CLOCK_MONOTONIC,&start);

                Zu messender Bereich

                clock_gettime(CLOCK_MONOTONIC,&finish);
                int secs = finish.tv_sec - start.tv_sec;
                int nsecs = abs(finish.tv_nsec - start.tv_nsec);
                printf("%i.%i Sekunden\n", secs, nsecs);
```

Abbildung 1: a = 12345 b = 20211 10.185750 Sekunden number of iterations: 10 gcd: 3

3.3.4 Zeitmessungen und Iterationskarten

In diesem Abschnitt finden Sie die Iterationskarten und die dazugehörigen Zeiten. Die jeweiligen Werte für die Iterationwerte sind ebenfalls mitangegeben.
Anmerkung: Da die Iterationskarten nur aus Graustufen bestehen sieht man hier leider nur einen grauen Block. Das liegt daran dass ich sehr hohe Werte wählen musste um die geforderte mindest Zeit von 10 Sekunden zu erreichen. Mit niedrigeren Werte entsteht an sinnvolles Diagramm. z.B 3.3.4 oder 3.3.4
3.3.4 wurde ohne eine Optimize Option erstellt und 3.3.4 wurde mit der Option -O3 erstellt.

3.4 numerische Integration

3.4.1 Aufbau und Kernfunktion

Ziel: Erstellen einer Struktur in der alle Variablen und Argumente für die numerische Integration abgelegt werden sollen. Passent dazu soll eine Funktion cmdargs_parse() erstellt, die genutzt wird, um die Struktur zu Begin des Programms zu initialisieren. Zudem soll die Funktion integral_sum() erstellt werden, die Variablen der zuvor erstellten Struktur übergeben bekommt und mit diesen Variablen das Integral einer Funktion f von x mit den Grenzen a und b berechnet.
Für die Ertstellung der Struktur habe ich die Vorlage aus der vorherigen Aufgabe verwendet und entsprechend die benötigten Variablen hinzugefügt oder verändert.

```
Untere Grenze für die Integration double a
obere Grenze für die Integration b
eine const char[] f welcher die Funktion speichern soll
```

Abbildung 2: Bei selben Werte 6.526717126 Sekunden

Abbildung 3: experiment 3

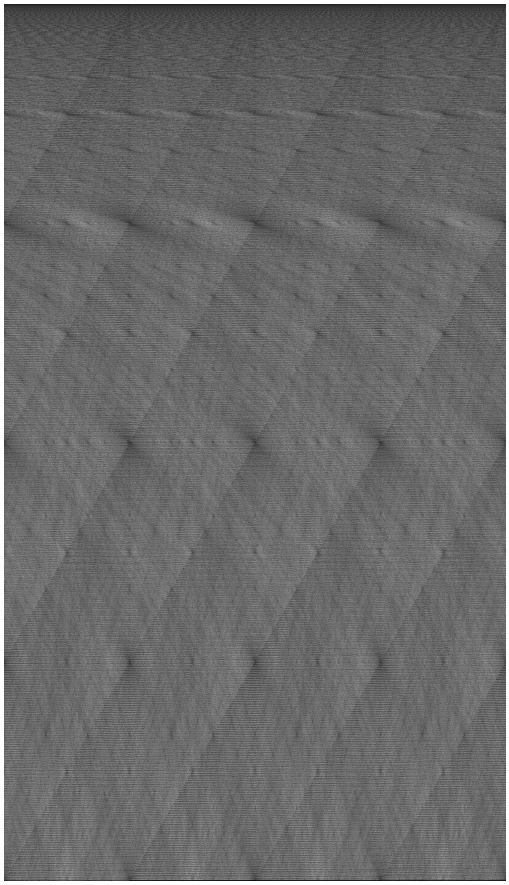
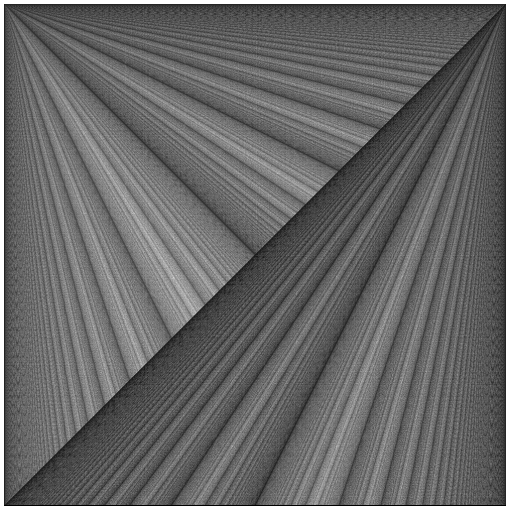


Abbildung 4: test



`int monte_carlo für die Fallunterscheidung, welcher Algorithmus verwendet werden soll.`
`unit64_t steps für die Anzahl der Schritte bei der Integration`
`monte_carlo und steps werden in Form von flags angegeben.`

Da cmdargs_parse() auch schon Teil der vorherigen Aufgabe war, habe ich den Code größtenteils kopiert und einige Änderungen bei den Fallunterscheidungen hinzu gefügt, sowie die in früheren Laboren erstellte Funktion argparse_int() verwendet, um die Verarbeitung der Argumente(Variablen Cast) zu verbessern. Die vollständigen Programme finden Sie in dem entsprechenden Repository, denn Code hier einzublenden würde den Rahmen sprengen. Für die numerische Integration habe ich mich für den 1/3 Simpson Algorithmus entschieden. Ich habe versucht die Algorithmen für numerische Integration anhand von Beispielen nicht niedrigen Schrittweiten nachzuvollziehen. Da mir der simpsonsche Algorithmus bei händischer Integration am besten gefiehlt habe ich dazu entschieden diesen auch zu nutzen.

```
double integral_sum(double (*f)(double x, void *userdata), void *userdata, double a,
double b, uint64_t steps){
double result, h;
        h = (b-a)/steps;
        result = f(a,userdata) + f(b,userdata);
        for(int i = 1; i <= steps-1;i++){
                double pos = a + i*h;
                if(i%2==0){
                        result += 2 * f(pos,userdata);
                }
                else {
                        result += 4 * f(pos,userdata);
                }
```

```
        }
        result *= h/3;
        return result;
}
```

Der Prototyp der Funktion war vorgegeben und ist auf dem Laborblatt.4. zufinden.

3.4.2 Testen der vorläufigen Funktion anhand einer Platzhalter Funktion

Platzhalter Funktion f(x)=2*sqrt(1-x*x) die folgenden Aufrufe habe diese Ergebnise zurückgebenen:
$ ./integrate ”2*sqrt(1-x*x)1 1 –steps 10000000
3.141592613 →korrekt
$ ./integrate ”2*sqrt(1-x*x)2 2
-nan Diese Fehlermeldung wird erzeugt, da negative Wurzeln einen mathematischen Fehler darstellen. →die Werte -2 bis 2 liegen auSSerhalb des Definitionsbereichs von f(x). Die Funktion ist korrekt.

3.4.3 Platzhalter durch eine Variable ersetzen

Ziel: Das Programm Integral soll nun nicht mehr mit einer Platzhalter Funktion funktionieren. Es soll dem Nutzer ermöglicht werden selber eine Funktion mit einer Variable anzugeben, welche dann für die Berechnung des Integral verwendet wird.
Dafür wird das Programm tinyexpr.c und die Header-Datei tinyexpr.h verwendet. Ich möchte hier anmerken, dass ich keine Rechte an dem tinyexpr Projekt habe. Dieses Projekt ist Open Scourse und darf von jedem benutzt werden, allerdings sollte man immer angeben wer dieses Code geschrieben hat. Ein Verweis auf die entsprechende Git-Lab Instanz ist im Literaturverzeichnis zufinden.
Im Laborblatt ist eine detaillierte Anleitung zufinden, wie tinyexpr in das Integral Projekt eingefügt werden kann. Daher kann ich nicht viel zum Vorgehen sagen, da ich mich strickt an die Anleitung gehalten habe. Mittels einer neuen Struktur soll jetzt eine Schnittstelle erzeugt werden, mit der man die Funktionen von tinyexpr nutzen kann, ohne die nötigen Hintergründe wie Variablen und so weiter jedes mal neu anzugeben. Das verbessert die Wiederverwendbarkeit des Code bezeihungsweise des Projekts. Die Struktur initialisiert alle nötigen Variablen, die für die Verwendung der tinyexpr Funktionen gebraucht werden. In einer Header-Datei sind die Struktur und die Schnittstellen Funktionen definiert und der entsprechenden Companion-Datei werden diese dann deklariert. Die hier wichtigen Funktionen sind te_complie(),te_free[] und te_eval(). Auch hier gab es eine Anleitung mit der ich allerdings einige Probleme hatte. Nur mithilfe des Dozenten konnte ich verstehen, wie die Zeigerfunktionen richtig zu schreiben und zu verwenden sind.

```
struct te_context{
        te_expr *expr;
        te_variable vars;
```

```
        double x;
};
int te_context_init(struct te_context *bind, const char *function){
        int err = 0;
        te_variable varr = {.name="x", .address=&bind->x};
        bind->vars = varr;
        bind->expr = te_compile(function,&bind->vars,1,&err);
        return err;
}

int te_context_deinit(struct te_context *bind){
        te_free(bind->expr);
        return 0;

}

double te_context_eval(struct te_context *bind, double x){
        bind->x = x;
        double fval = te_eval(bind->expr);
        return fval;
}
```

AbschlieSSend werden diese Funktionen mit einem Cast nutzbar gemacht. Bei diesem Cast wird eine bereits initialisierte Struktur an eine neue Funktion (f) übergeben. Diese erwartet einen double Wert und void Zeiger. Der Zeiger wird zu der Struktur gecastet, die die Schnittstellen Funktionen enthält. Diese Funktion wird ebenfalls als Zeiger an die Kernfunktion integral_sum() übergeben, weil das Integrtal in dieser Funktion berechnet wird und nicht im Main Programm.

3.4.4 Monte-carlo und Zeitmessungen

Ziel: Implementierung eines zusätzlichen Integrationsverfahren (Monte-Carlo) und sowie Möglichkeit die Zeit der numerischen Integration zu messen. Wie in der Voraufgabe habe ich hier die selbe Methode benutzt. Ich habe den zu messenden Bereich mit der Funktion clock_gettime umschlossen. Werte für die Zeit werden dann auf der Konsole ausgegeben. Eine genaue Beschreibung, wie der Monte-carlo Algorithmus funktioniert ist auf dem Laborblatt.4. auf Folie 11 beziehungsweise 12. Ich möchte hier nur kurz auf meine Implemenation eingehen. Der Prototyp der Funktion für den M-C Algorithmus ist identisch zum Prototypen der Funktion integral_sum(), da die neue Funktion integral_mc() die selben Variablen benötigt. Der Kern der Funktion besteht daraus, dass ein Integral für eine zufällige Zahl, die zwischen den Grenzen liegt, berechnet wird. dafür benutze ich die Funktion rand(), welche in der C-Standardbibliothek math.h enthalten ist. Es wird pro step ein solcher Wert erstellt. Die Integrationswerte werden allen zusammen addiert und abschlieSSend mit dem Bruch (obere Grenze - untere Grenze) geteilt

durch die Anzahl der Schritte geteilt. Das Ergebnis dieser Rechnung wird anschlieSSend zurückgegeben.

```
double integral_mc(double (*f)(double x, void *userdata), void *userdata, double a,
double b, uint64_t steps){
        double result, rndomnum;
        double val = 0.0;

        for(int i = 0; i < steps-1; i++){
                rndomnum = a+((double)rand()/RAND_MAX) * (b-a);
                val += f(rndomnum,userdata);

        }
        result = (b-a)*val/steps;
        return result;
}
```

Damit im Hauptteil des Main Programms keine Fallunterscheidung, welcher Integrationsalgorithmus angewendet werden soll, stehen muss benutze ich eine Funktionm, die überprüft ob die entsprechende flag (–monte-carlo) gegeben ist. Innerhalb dieser Funktion wird dann der entsprechende Algorithmus aufgerufen und das Ergebnis wird direkt auf der Konsole ausgegeben. Deshalb hat auch die Funktion den selben Prototyp wie integral_sum() und integral_mc(). Nur mit dem Unterschied, dass noch eine Integer Variable mit übergeben wird. Dieser Integer gibt an, welcher Algorithmus von Nutzer gewünscht ist.

```
void integrate(double (*f)(double x, void *userdata), void *userdata, double a,
double b, uint64_t steps, int monte){
        if(monte == 1){
                double flaecheninhalt = integral_mc(f,userdata,a,b,steps);
                printf("%.9f\n", flaecheninhalt);
                te_context_deinit(userdata);
        }
        else{
                double flaecheninhalt = integral_sum(f,userdata,a,b,steps);
                printf("%.9f\n", flaecheninhalt);
                te_context_deinit(userdata);

        }

}
```

3.5 sdl2 erste Schritte

Ziel: Anpassung eines Beispielprogramms, welches sdl2 nutzt. Konkret sollte ein Quadrat zurückgesetzt werden, wenn es den sichtbaren Bereich verlässt. Das Quadrat bewegt sich

wie in einem Koordinatensystem mit einer gewissen Geschwindigkeit in x und y Richtung, wobei die Geschwindigkeiten nicht identisch sind. Am Ende jedes Funktionsaufrufen wird die y-Geschwindigkeit um 1 inkrementiert. Dieser Vorgang und die Lösung sind in der Funktion game_state_update geschrieben. Ich habe das Programm erstmal ausgeführt, um zu überprüfen, wann das Quadrat den sichtbaren Bereich verlässt. Ich bin zu dem Schluss gekommen, dass das Quadrat ungefähr bei der x-Koordinate 650 nicht mehr zusehen ist. Dem entsprechend habe ich eine Fallunterscheidung eingefügt, die überprüft welchen Wert die x-Koordinate hat. ist die größer oder gleich 650 soll die x-Koordinate wieder auf 1 zurückgesetzt werden. Dabei hatte ich allerdings ein Problem nicht bedacht und zwar, dass die y-Geschwindigkeit immer weiter inkrementiert wird. Wenn sich das Quadrat zu schnell in y-Richtung bewegt, kommt es zu einem Glitch. Als Folge dessen verschwindet das Quadrat jetzt in y-Richtung aus dem sichtbaren Bereich. Um das zu verhindern müssen auch die y-Koordinate und die y-Geschwindigkeit weider auf ihren Ursprungswert zurückgesetzt werden.

```
void game_state_update(struct game_state *state)
{
    state->x += state->speedx;
    state->y += state->speedy;

    if (state->y >= 400) {
        state->speedy *= -1;
    }
        /Mein Code/
        if(state->x >= 642){
                state->x = 1;
                state->y = 50;
                state->speedy = 1;
        }

    state->speedy += 1;
}
```

Dieser Code sorgt für eine reibungslosen Ablauf und es kommt zu keinen Störungen. Zusätzlich habe ich die Überprüfung der x-Koordinate noch etwas angepasst, wodurch die Verzögerung vor dem reset für das menschliche Auge so kurz wie möglich ist.

4 Fazit

Ich habe während der Bearbeitung des 4. Laborblattes festgestellt, wie nützlich und hilfreich und effizient ein auf Wiederverwendbarkeit ausgelegter Programm Aufabu ist. Besonders bei Makefiles spielt ein solcher Aufbau eine große Rolle. Wiederverwendbarkeit kann die Arbeit in C drastisch vereinfachen. Daher finde ich es sehr gut, dass im

Rahmen des Labors der Fokus auch auf solche Teilaspekt des Programmierens gelegt wird. Außerdem musste ich zu meinem Leidwesen feststellen, dass meine bisherigen Erfahrungen und mein Wissen im Bereich Programmieren allein nicht mehr ausreichen. Ich hatte mich zu sehr auf mein Wissen mit der recht ähnlichen Programmiersprache Java verlassen und habe dadurch unnötige Fehler gemacht, die meinen Fortschritt stark beeinflusst haben. Das habe ich besonders bei Zeigern und Strukturen gemerkt. Daher war dieses Labor für mich eine wichtige Erfahrung, die mir meine Schwächen aufgezeigt hat und somit eine Möglichkeit bietet, mich als Programmierer zu verbessern. AbschlieSSend möchte ich noch anmerken, dass mein größtes Problem in diesem Labor die Zeit war. Wie auch schon im letzten Labor war die Zeit sehr knapp. Als Folge dessen konnte die Abgabe nicht rechtzeitig getätigt werden. Anders als bei letztem Labor bin ich allerdings der Ansicht, dass dies nicht an dem Laborumfang lag, sondern an meiner Arbeitsweise liegt. Ich habe bisher alle Aufgaben, die eigentlich als Gruppenaufgabe gedacht waren, strickt allein gearbeitet, da ich persönlich glaube, dass man nur durch Einzelarbeit richtig lernen kann. Hätte ich nur eine der beiden Hauptaufgaben bearbeiten müssen und einige andere Arbeiten nach hinten verschoben. Wäre ich höchstwahrscheinlich in der Lage gewesen, das Labor rechtzeitig vor der Weihnachtspause abzuschließen. Allerdings bin ich auch der Meinung, dass ich nur durch Einzelarbeit in der Lage war, meine eigenen Schwächen zu identifizieren. Als persönliches Fazit kann ich aus dem Labor mitnehmen, dass ich nicht alles allein machen kann, wenn ich im vorgesehenen Zeitrahmen bleiben möchte. Auch wenn das gegen meine bevorzugte Arbeitsweise spricht. Die Frage ist jetzt, was das Ziel eines Studiums beziehungsweise eines Moduls ist. Alle Aufgaben vorschriftsgemäSS rechtzeitig einzureichen oder etwas für die Zukunft und den gewünschten Berufsweg zu lernen. Alle Dateien und Programme dieses Labors sind in meinem Git-Lab Repository zu finden.

5 Literaturverzeichnis

Vorlesungen des Moduls
Stack Overflow comments in Markdown
EARTHLY Variables in Makefile
gentoo linux
MathWorks
GeeksforGeeks GNU Preprocessors GNU including Makefiles GNU Prerequisites GNU File Names GNU Variables Simpson Algorithmus Monte-Carlo Algorithmus tinyexpr
Git-Lab Instanz bitte anschauen