```
.
├── 4by3.pdf
├── A4.pdf
├── benchmark
│   └── polynomial_approximation
│       ├── benchmark_gradlen.sh
│       ├── benchmark_parallel.sh
│       ├── benchmark_reverse.sh
│       ├── benchmark_workers.sh
│       ├── benchmark.sh
│       ├── forward_parallel.cpp
│       ├── forward.cpp
│       ├── Makefile
│       ├── primal.cpp
│       └── reverse.cpp
├── example
│   ├── hello_world
│   │   ├── forward
│   │   ├── forward.cpp
│   │   ├── Makefile
│   │   ├── reverse
│   │   └── reverse.cpp
│   └── polynomial_approximation
│       ├── forward
│       ├── forward_parallel
│       ├── forward_parallel.cpp
│       ├── forward.cpp
│       ├── Makefile
│       ├── plot.py
│       ├── reverse
│       └── reverse.cpp
├── forward.h
├── generate-pdf.sh
├── LICENSE
├── README.md
├── reverse.h
└── tmp-pdf
    └── 4by3-0.pdf

7 directories, 31 files
```

```
/*
 * ============================================================================
 * Simple Tape-Based Reverse Mode Autodiff
 * ============================================================================
 * This header-only C implementation provides reverse mode automatic
 * differentiation using a dynamic computation tape and operator overloading
 * on a custom `var_t` type.
 *
 * Each `var_t` variable corresponds to a node on a global tape. The tape
 * records the computation graph by tracking the operation and parent variables
 * for each intermediate result.
 *
 * After constructing a function from input `var_t`s, a reverse pass propagates
 * gradients from the output node backward through the tape using the chain
 * rule.
 *
 * Usage Example:
 * ----------------------------------------------------------------------------
 * To compute ∂f/∂x and ∂f/∂y for f(x, y) = sin(x) + y²:
 *   tape_t tape = tape_create(64);
 *   tape_load(tape);
 *   var_t x = var_create(1.0f);
 *   var_t y = var_create(2.0f);
 *   var_t f = var_sin(x) + var_pow(y, var_create(2.0f));
 *   tape_reverse_pass(tape, f);
 *   // var_adjoint(x) returns ∂f/∂x, var_adjoint(y) returns ∂f/∂y
 *
 * Notes:
 * ----------------------------------------------------------------------------
 *  - Always call `tape_load()` before creating variables.
 */

#ifndef H_AUTODIFF
#define H_AUTODIFF

#include <stddef.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <string.h>

const uint32_t MAX_TAPE_LENGTH = 1 << 24;  /* correspond to a ~330mb tap */

typedef enum {
  NIL = 0,
  NEG,
  ADD,
  SUB,
  MUL,
  DIV,
  POW,
  EXP,
  COS,
  SIN,
  SQRT,
} operator_t;

typedef struct {
  float value;
  float adjoint;
  uint32_t left_parent;
  uint32_t right_parent;
  operator_t op;
} tape_entry_t;

typedef struct {
  uint32_t length;
  uint32_t capacity;
  tape_entry_t *entries;
} tape_t;

typedef struct {
  uint32_t index;
} var_t;

/* should not be set directly, use `tape_load` instead */
static tape_t global_tape = {
  .length = 0,
  .capacity = 0,
  .entries = NULL,
};
```

```
/*
 * setting the initial capacity of the tape to a number like 64 will prevent too
 * much calls to realloc
 */
static tape_t tape_create(uint32_t capacity) {
  assert(capacity <= MAX_TAPE_LENGTH);
  tape_entry_t *entries = (tape_entry_t *) calloc(capacity, sizeof(tape_entry_t));
  if (entries == NULL) {
    perror("tape malloc");
    exit(1);
    return {};
  }
  return {
    .length = 0,
    .capacity = capacity,
    .entries = entries,
  };
}

static void tape_destroy(tape_t tape) {
  free(tape.entries);
}

static void tape_extend(tape_t tape) {
  assert(tape.length < MAX_TAPE_LENGTH);
  if (tape.length == tape.capacity) {
    tape.entries = (tape_entry_t *) realloc(tape.entries, 2 * tape.capacity * sizeof(*tape.entries));
    if (tape.entries == NULL) {
      perror("tape realloc");
      exit(1);
      return;
    }
    memset(tape.entries + tape.capacity, 0, tape.capacity * sizeof(*tape.entries));
    tape.capacity = 2 * tape.capacity;
  }
  ++tape.length;
}

static void tape_clear(tape_t tape) {
  memset(tape.entries, 0, tape.length * sizeof(*tape.entries));
  tape.length = 0;
}

static void tape_load(tape_t tape) {
  global_tape = tape;
}

static tape_t tape_loaded() {
  return global_tape;
}

static void tape_reverse_pass(tape_t tape, var_t start) {
  for (size_t i = 0; i < tape.length; ++i)
    tape.entries[i].adjoint = 0;
  tape.entries[start.index].adjoint = 1;

  for (size_t i = start.index+1; i-- > 0;) {  /* avoid size_t wraps */
    tape_entry_t *entry = &tape.entries[i];
    tape_entry_t *left_parent_entry = &tape.entries[entry->left_parent];
    tape_entry_t *right_parent_entry = &tape.entries[entry->right_parent];
    switch (entry->op) {
      case NIL:
        break;
      case NEG:
        left_parent_entry->adjoint += entry->adjoint * -1;
        break;
      case ADD:
        left_parent_entry->adjoint  += entry->adjoint * 1;
        right_parent_entry->adjoint += entry->adjoint * 1;
        break;
      case SUB:
        left_parent_entry->adjoint  += entry->adjoint * 1;
        right_parent_entry->adjoint += entry->adjoint * -1;
        break;
      case MUL:
        left_parent_entry->adjoint  += entry->adjoint * right_parent_entry->value;
        right_parent_entry->adjoint += entry->adjoint * left_parent_entry->value;
        break;
      case DIV:
        left_parent_entry->adjoint  += entry->adjoint / right_parent_entry->value;
```

```
        right_parent_entry->adjoint += entry->adjoint * -1 * (entry->value / right_parent_entry->value);
        break;
      case POW:
        left_parent_entry->adjoint  += entry->adjoint * right_parent_entry->value * (entry->value /
left_parent_entry->value);
        right_parent_entry->adjoint += entry->adjoint * entry->value * logf(left_parent_entry->value);
        break;
      case EXP:
        left_parent_entry->adjoint += entry->adjoint * entry->value;
        break;
      case COS:
        left_parent_entry->adjoint += entry->adjoint * -1 * sqrtf(1 - entry->value*entry->value);
        break;
      case SIN:
        left_parent_entry->adjoint += entry->adjoint * sqrtf(1 - entry->value*entry->value);
        break;
      case SQRT:
        left_parent_entry->adjoint += entry->adjoint / (2 * entry->value);
        break;
    }
  }
}

/* append new variable to global_tape */
static var_t var_create(float value) {
  var_t a = {global_tape.length};
  tape_extend(global_tape);
  global_tape.entries[a.index].value = value;
  return a;
}

static float var_adjoint(var_t a) {
  return global_tape.entries[a.index].adjoint;
}

static float var_value(var_t a) {
  return global_tape.entries[a.index].value;
}

/* variable operations */
static var_t operator-(var_t a) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  var_t b = var_create(-a_entry->value);
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  b_entry->op = NEG;
  b_entry->left_parent = a.index;
  return b;
}


/* variable variable operations */
static var_t operator+(var_t a, var_t b) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  var_t c = var_create(a_entry->value + b_entry->value);
  tape_entry_t *c_entry = &global_tape.entries[c.index];
  c_entry->op = ADD;
  c_entry->left_parent = a.index;
  c_entry->right_parent = b.index;
  return c;
}

static var_t operator-(var_t a, var_t b) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  var_t c = var_create(a_entry->value - b_entry->value);
  tape_entry_t *c_entry = &global_tape.entries[c.index];
  c_entry->op = SUB;
  c_entry->left_parent = a.index;
  c_entry->right_parent = b.index;
  return c;
}

static var_t operator*(var_t a, var_t b) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  var_t c = var_create(a_entry->value * b_entry->value);
  tape_entry_t *c_entry = &global_tape.entries[c.index];
  c_entry->op = MUL;
  c_entry->left_parent = a.index;
  c_entry->right_parent = b.index;
```

```
  return c;
}

static var_t operator/(var_t a, var_t b) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  var_t c = var_create(a_entry->value / b_entry->value);
  assert(b_entry->value != 0);
  tape_entry_t *c_entry = &global_tape.entries[c.index];
  c_entry->op = DIV;
  c_entry->left_parent = a.index;
  c_entry->right_parent = b.index;
  return c;
}

static void operator+=(var_t &a, var_t b) {
  a = a + b;
}

static void operator-=(var_t &a, var_t b) {
  a = a - b;
}

static void operator*=(var_t &a, var_t b) {
  a = a * b;
}

static void operator/=(var_t &a, var_t b) {
  a = a / b;
}

/* variable functions */
static var_t var_pow(var_t a, var_t b) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  assert(a_entry->value > 0);
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  var_t c = var_create(powf(a_entry->value, b_entry->value));
  tape_entry_t *c_entry = &global_tape.entries[c.index];
  c_entry->op = POW;
  c_entry->left_parent = a.index;
  c_entry->right_parent = b.index;
  return c;
}

static var_t var_exp(var_t a) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  var_t b = var_create(expf(a_entry->value));
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  b_entry->op = EXP;
  b_entry->left_parent = a.index;
  return b;
}

static var_t var_cos(var_t a) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  var_t b = var_create(cosf(a_entry->value));
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  b_entry->op = COS;
  b_entry->left_parent = a.index;
  return b;
}

static var_t var_sin(var_t a) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  var_t b = var_create(sinf(a_entry->value));
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  b_entry->op = SIN;
  b_entry->left_parent = a.index;
  return b;
}

static var_t var_sqrt(var_t a) {
  tape_entry_t *a_entry = &global_tape.entries[a.index];
  /* assert(a_entry->value > 0); */
  var_t b = var_create(sqrtf(a_entry->value));
  tape_entry_t *b_entry = &global_tape.entries[b.index];
  b_entry->op = SQRT;
  b_entry->left_parent = a.index;
  return b;
}
```

```
#endif
```

```
/*
 * ============================================================================
 * Simple Vectorized Forward Mode Autodiff
 * ============================================================================
 * This header-only C implementation provides forward mode automatic
 * differentiation using operator overloading on a custom `var_t` type.
 *
 * Each `var_t` variable holds:
 *  - `value`: the scalar value of the variable.
 *  - `grad[GRADLEN]`: a gradient vector representing the derivative of the
 *    variable with respect to each input in a vector of size `GRADLEN`.
 *
 * Usage Example:
 * ----------------------------------------------------------------------------
 * To compute ∂f/∂x and ∂f/∂y for f(x, y) = sin(x) + y²:
 *   var_t x = {.value = 1.0}; x.grad[0] = 1; // ∂x/∂x = 1
 *   var_t y = {.value = 2.0}; y.grad[1] = 1; // ∂y/∂y = 1
 *   var_t f = var_sin(x) + var_pow(y, 2);
 *   // f.value holds the result, f.grad[0] is ∂f/∂x, f.grad[1] is ∂f/∂y
 *
 * Notes:
 * ----------------------------------------------------------------------------
 *  - The macro `GRADLEN` must be defined before including this header.
 */

#ifndef H_AUTODIFF
#define H_AUTODIFF

#include <stddef.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <string.h>

/* gradient length */
#ifndef GRADLEN
#error "The GRADLEN macro must set before including fowrard.h"
#define GRADLEN 0
#endif

typedef struct {
  float grad[GRADLEN];
  float value;
} var_t;

/*
 * initialize an new variable that does not derive from the input vector (see
 * above description)
 */
static void var_zero(var_t *a) {
  memset(a, 0, sizeof(*a));
}

/* variable operations */
static var_t operator-(var_t a) {
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = -a.grad[i];
  a.value = -a.value;
  return a;
}

/* variable variable operations */
static var_t operator+(var_t a, const var_t &b) {
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = a.grad[i] + b.grad[i];
  a.value = a.value + b.value;
  return a;
}

static var_t operator-(var_t a, const var_t &b) {
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = a.grad[i] - b.grad[i];
  a.value = a.value - b.value;
  return a;
}

static var_t operator*(var_t a, const var_t &b) {
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = b.value * a.grad[i] + a.value * b.grad[i];
  a.value = a.value * b.value;
  return a;
```

```
}

static var_t operator/(var_t a, const var_t &b) {
  assert(b.value != 0);
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = (b.value * a.grad[i] - a.value * b.grad[i]) / (b.value * b.value);
  a.value = a.value / b.value;
  return a;
}

static void operator+=(var_t &a, const var_t &b) {
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = a.grad[i] + b.grad[i];
  a.value = a.value + b.value;
}

static void operator-=(var_t &a, const var_t &b) {
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = a.grad[i] - b.grad[i];
  a.value = a.value - b.value;
}

static void operator*=(var_t &a, const var_t &b) {
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = b.value * a.grad[i] + a.value * b.grad[i];
  a.value = a.value * b.value;
}

static void operator/=(var_t &a, const var_t &b) {
  assert(b.value != 0);
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = (b.value * a.grad[i] - a.value * b.grad[i]) / (b.value * b.value);
  a.value = a.value / b.value;
}

/* variable float operations */
static var_t operator+(var_t a, float b) {
  a.value += b;
  return a;
}

static var_t operator-(var_t a, float b) {
  a.value -= b;
  return a;
}

static var_t operator*(var_t a, float b) {
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] *= b;
  a.value *= b;
  return a;
}

static var_t operator/(float a, var_t b) {
  for (size_t i = 0; i < GRADLEN; i++)
    b.grad[i] = -a * b.grad[i] / (b.value * b.value);
  b.value = a / b.value;
  return b;
}

static void operator+=(var_t &a, float b) {
  a.value += b;
}

static void operator-=(var_t &a, float b) {
  a.value -= b;
}

static void operator*=(var_t &a, float b) {
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] *= b;
  a.value *= b;
}

static void operator/=(float a, var_t &b) {
  for (size_t i = 0; i < GRADLEN; i++)
    b.grad[i] = -a * b.grad[i] / (b.value * b.value);
  b.value = a / b.value;
}

/* variable functions */
```

```
static var_t var_pow(var_t a, float b) {
  assert(a.value > 0);
  float pow = powf(a.value, b-1);
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = b * a.grad[i] * pow;
  a.value = powf(a.value, b);
  return a;
}

static var_t var_exp(var_t a) {
  float expa = expf(a.value);
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = a.grad[i] * expa;
  a.value = expa;
  return a;
}

static var_t var_cos(var_t a) {
  float sina = -sinf(a.value);
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = a.grad[i] * sina;
  a.value = cosf(a.value);
  return a;
}

static var_t var_sin(var_t a) {
  float cosa = cosf(a.value);
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = a.grad[i] * cosa;
  a.value = sinf(a.value);
  return a;
}

static var_t var_sqrt(var_t a) {
  /* assert(a.value > 0); */
  for (size_t i = 0; i < GRADLEN; i++)
    a.grad[i] = 0.5 * a.grad[i] / sqrtf(a.value);
  a.value = sqrtf(a.value);
  return a;
}

#endif
```

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <math.h>
#include <time.h>

const int N = 1000;  /* number of terms in the reimann sum */
const int DEG = 10;  /* degree of the polynomial proximation */
const float START = 0;  /* the start of the integration interval */
const float END = 2;  /* the end of the integration interval */
const float ITERATIONS = 5000;  /* number of gradient descent iterations */
const float ALPHA = 0.001;  /* gradient descent speed */

#include "../../reverse.h"

/* the function to approximate */
float f(float x) {
  if (x == 0) return 0;
  return exp(-1 / (x*x));
}

var_t poly_eval(var_t P[DEG+1], float x) {
  var_t val = P[0];
  float X = x;
  for (size_t i = 1; i < DEG+1; i++) {
    val += P[i] * var_create(X);
    X *= x;
  }
  return val;
}

void poly_init(var_t P[DEG+1]) {
  for (size_t i = 0; i < DEG+1; ++i) {
    P[i] = var_create(i+1);
  }
}

void poly_print(float P[DEG+1]) {
  printf("polynomial: ");
  for (size_t i = 0; i < DEG; ++i) {
    printf("%f, ", P[i]);
  }
  printf("%f\n", P[DEG]);
}

var_t reimann_integral(var_t P[DEG+1]) {
  var_t loss = {0};

  float step_size = (END-START) / N;
  for (size_t j = 0; j < N; ++j) {
    float x = START + j*step_size;
    var_t delta = poly_eval(P, x) - var_create(f(x));
    loss = loss + (delta*delta) * var_create(step_size);
  }

  return loss;
}

void polynomial_approximation(float P_coef[DEG+1]) {
  tape_t tape = tape_create(64);
  tape_load(tape);

  var_t P[DEG+1];
  poly_init(P);

  for (size_t i = 0; i < ITERATIONS; ++i) {
    /* reimann integral */
    var_t loss = reimann_integral(P);
    /* printf("loss: %f\n", var_value(loss)); */

    /* gradient descent */
    tape_reverse_pass(tape, loss);
    for (size_t j = 0; j < DEG+1; ++j) {
      /* normalize the influance of X^j */
      float one_over_norm_of_xj = j / (powf(END, j+1) - powf(START, j+1));
      P_coef[j] = var_value(P[j]) - ALPHA * var_adjoint(P[j]) * one_over_norm_of_xj;
    }

    /* update polynomial */
    tape_clear(tape);
    for (size_t j = 0; j < DEG+1; ++j) {
```

```
      P[j] = var_create(P_coef[j]);
    }
  }

  tape_destroy(tape);
}

int main() {
  float P[DEG+1];
  polynomial_approximation(P);
  poly_print(P);

  return 0;
}
```

```
all: build

CC := clang
CFLAGS := -std=c++11 -O2 -lm

build: forward.cpp reverse.cpp forward_parallel.cpp
	$(CC) $(CFLAGS) forward.cpp -o forward
	$(CC) $(CFLAGS) reverse.cpp -o reverse
	$(CC) $(CFLAGS) -pthread forward_parallel.cpp -o forward_parallel

dev: forward.cpp reverse.cpp forward_parallel.cpp
	$(CC) -std=c++11 -g -lm forward.cpp -o forward
	$(CC) -std=c++11 -g -lm reverse.cpp -o reverse
	$(CC) -std=c++11 -g -lm -pthread forward_parallel.cpp -o forward_parallel

clean:
	rm -rf forward reverse
```

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <math.h>
#include <time.h>

const int N = 1000;  /* number of terms in the reimann sum */
const int DEG = 10;  /* degree of the polynomial proximation */
const float START = 0;  /* the start of the integration interval */
const float END = 2;  /* the end of the integration interval */
const float ITERATIONS = 5000;  /* number of gradient descent iterations */
const float ALPHA = 0.001;  /* gradient descent speed */

#define GRADLEN (DEG+1)
#include "../../forward.h"

/* the function to approximate */
float f(float x) {
  if (x == 0) return 0;
  return exp(-1 / (x*x));
}

var_t poly_eval(var_t P[DEG+1], float x) {
  var_t val = P[0];
  float X = x;
  for (size_t i = 1; i < DEG+1; i++) {
    val += P[i] * X;
    X *= x;
  }
  return val;
}

void poly_init(var_t P[DEG+1], size_t grad_start, size_t grad_end) {
  for (size_t i = 0; i < DEG+1; ++i) {
    P[i] = {.grad = {0}, .value = (float) i+1};
    if (i >= grad_start && i < grad_end) {
      P[i].grad[i - grad_start] = 1;
    }
  }
}

void poly_print(float P[DEG+1]) {
  printf("polynomial: ");
  for (size_t i = 0; i < DEG; ++i) {
    printf("%f, ", P[i]);
  }
  printf("%f\n", P[DEG]);
}

var_t reimann_integral(var_t P[DEG+1]) {
  var_t loss = {0};

  float step_size = (END-START) / N;
  for (size_t j = 0; j < N; ++j) {
    float x = START + j*step_size;
    var_t delta = poly_eval(P, x) - f(x);
    loss = loss + (delta*delta) * step_size;
  }

  return loss;
}

void polynomial_approximation(float P_coef[DEG+1]) {
  var_t P[DEG+1];
  poly_init(P, 0, GRADLEN);

  for (size_t i = 0; i < ITERATIONS; ++i) {
    /* reimann integral */
    var_t loss = reimann_integral(P);
    /* printf("loss: %f\n", loss.value); */

    /* gradient descent */
    for (size_t j = 0; j < DEG+1; ++j) {
      float one_over_norm_of_xj = j / (powf(END, j+1) - powf(START, j+1));
      /* normalize the influance of X^i */
      P[j].value -= ALPHA * loss.grad[j] * one_over_norm_of_xj;
    }
  }

  for (size_t i = 0; i < DEG+1; ++i) {
    P_coef[i] = P[i].value;
```

```
  }
}

int main() {
  float P[DEG+1];
  polynomial_approximation(P);
  poly_print(P);

  return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <math.h>
#include <time.h>
#include <pthread.h>
#include <assert.h>

const int N = 1000;  /* number of terms in the reimann sum */
const int DEG = 10;  /* degree of the polynomial proximation */
const float START = 0;  /* the start of the integration interval */
const float END = 2;  /* the end of the integration interval */
const float ITERATIONS = 5000;  /* number of gradient descent iterations */
const float ALPHA = 0.001;  /* gradient descent speed */

#define GRADLEN 32
#include "../../forward.h"

/* the function to approximate */
float f(float x) {
  if (x == 0) return 0;
  return exp(-1 / (x*x));
}

var_t poly_eval(var_t P[DEG+1], float x) {
  var_t val = P[0];
  float X = x;
  for (size_t i = 1; i < DEG+1; i++) {
    val += P[i] * X;
    X *= x;
  }
  return val;
}

void poly_print(float P[DEG+1]) {
  printf("polynomial: ");
  for (size_t i = 0; i < DEG; ++i) {
    printf("%f, ", P[i]);
  }
  printf("%f\n", P[DEG]);
}

#define RI_WORKERS 2
const size_t RI_CHUNKS = ((DEG+1 + GRADLEN-1) / GRADLEN);

typedef struct {
  size_t start_chunk;
  size_t end_chunk;
  float *P;
  float *grad;
  float value;
} ri_worker_param_t;

void *ri_worker(void *param_ptr) {
  ri_worker_param_t *param = (ri_worker_param_t *) param_ptr;

  for (size_t chunk_id = param->start_chunk; chunk_id < param->end_chunk; ++chunk_id) {
    var_t loss = {0};
    var_t P[DEG+1] = {0};
    for (size_t i = 0; i < DEG+1; ++i) {
      P[i].value = param->P[i];
      if (i >= chunk_id * GRADLEN && i < chunk_id * GRADLEN + GRADLEN) {
        P[i].grad[i - chunk_id * GRADLEN] = 1;
      }
    }

    float step_size = (END-START) / N;
    for (size_t j = 0; j < N; ++j) {
      float x = START + j*step_size;
      var_t delta = poly_eval(P, x) - f(x);
      loss = loss + (delta*delta) * step_size;
      if (loss.value != loss.value) {

      }
    }

    if (chunk_id == param->start_chunk) {
      param->value = loss.value;
    } else {
      /* assert(param->value == loss.value); */
    }
```

```
    for (size_t i = 0; i < GRADLEN && chunk_id * GRADLEN + i < DEG+1; ++i) {
      param->grad[chunk_id * GRADLEN + i] = loss.grad[i];
    }
  }

  return NULL;
}

float reimann_integral(float P[DEG+1], float grad[DEG+1]) {
  pthread_t worker_threads[RI_WORKERS];
  ri_worker_param_t worker_params[RI_WORKERS];

  assert(RI_CHUNKS > 0);
  int handled_chunks = 0;
  for (size_t worker_id = 0; worker_id < RI_WORKERS; ++worker_id) {
    size_t start_chunk = (size_t) ((float) RI_CHUNKS / RI_WORKERS * worker_id);
    size_t end_chunk = (size_t) ((float) RI_CHUNKS / RI_WORKERS * (worker_id+1));
    worker_params[worker_id] = {
      .start_chunk = start_chunk,
      .end_chunk = end_chunk,
      .P = P,
      .grad = grad,
    };
    int err = pthread_create(&worker_threads[worker_id], NULL, &ri_worker, &worker_params[worker_id]);
    if (err) {
      printf("pthread_create error %d", err);
      exit(1);
      return 0;
    }
    handled_chunks += end_chunk - start_chunk;
  }
  assert(handled_chunks == RI_CHUNKS);

  for (size_t worker_id = 0; worker_id < RI_WORKERS; ++worker_id) {
    int err = pthread_join(worker_threads[worker_id], NULL);
    if (err) {
      printf("pthread_join error %d", err);
      exit(1);
      return err;
    }
  }

  float value;
  for (size_t worker_id = 1; worker_id < RI_WORKERS; ++worker_id) {
    if (worker_params[worker_id].end_chunk - worker_params[worker_id].start_chunk > 0) {
      value = worker_params[worker_id].value;
    }
  }
  return value;
}

void polynomial_approximation(float P[DEG+1]) {
  for (size_t i = 0; i < DEG+1; ++i) {
    P[i] = i+1;
  }

  for (size_t i = 0; i < ITERATIONS; ++i) {
    /* reimann integral */
    float loss_grad[DEG+1];
    float loss = reimann_integral(P, loss_grad);
    /* printf("loss: %f\n", loss); */

    /* gradient descent */
    for (size_t j = 0; j < DEG+1; ++j) {
      float one_over_norm_of_xj = j / (powf(END, j+1) - powf(START, j+1));
      /* normalize the influance of X^i */
      P[j] -= ALPHA * loss_grad[j] * one_over_norm_of_xj;
    }
  }
}

int main() {
  float P[DEG+1];
  polynomial_approximation(P);
  poly_print(P);

  return 0;
}
```

```
#include <stdio.h>
#include "../../reverse.h"

int main() {
  tape_t tape = tape_create(64);
  tape_load(tape);

  var_t a = var_create(4);
  var_t b = var_create(9);
  var_t c = var_create(7);
  var_t d = var_create(-2);

  var_t e = var_pow(var_sqrt(a / (b + c * a) + var_exp(var_create(1) / d)), -var_create(3));
  tape_reverse_pass(tape, e);
  printf("value: %f\n", var_value(e));
  printf("grad: {%f, %f, %f, %f}\n", var_adjoint(a), var_adjoint(b), var_adjoint(c), var_adjoint(d));

  tape_destroy(tape);
  return 0;
}
```

```
all: build

CC := clang
CFLAGS := -std=c++11 -O2 -lm

build: forward.cpp reverse.cpp
	$(CC) $(CFLAGS) forward.cpp -o forward
	$(CC) $(CFLAGS) reverse.cpp -o reverse

clean:
	rm -rf forward reverse
```

```
#include <stdio.h>

#define GRADLEN 4
#include "../../forward.h"

int main() {
  var_t a = {.grad = {1, 0, 0, 0}, .value = 4};
  var_t b = {.grad = {0, 1, 0, 0}, .value = 9};
  var_t c = {.grad = {0, 0, 1, 0}, .value = 7};
  var_t d = {.grad = {0, 0, 0, 1}, .value = -2};

  var_t e = var_pow(var_sqrt(a / (b + c * a) + var_exp(1 / d)), -3);
  printf("value: %f\n", e.value);
  printf("grad: {%f, %f, %f, %f}\n", e.grad[0], e.grad[1], e.grad[2], e.grad[3]);

  return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <math.h>
#include <time.h>

const int N = 1000;  /* number of terms in the reimann sum */
const float START = 0;  /* the start of the integration interval */
const float END = 2;  /* the end of the integration interval */
#ifndef DEG
#warning "DEG set to default value 4"
const int DEG = 4;  /* degree of the polynomial proximation */
#endif

#include "../../reverse.h"

/* the function to approximate */
float f(float x) {
  if (x == 0) return 0;
  return exp(-1 / (x*x));
}

var_t poly_eval(var_t P[DEG+1], float x) {
  var_t val = P[0];
  float X = x;
  for (size_t i = 1; i < DEG+1; i++) {
    val = val + P[i] * var_create(X);
    X *= x;
  }
  return val;
}

void poly_init(var_t P[DEG+1]) {
  for (size_t i = 0; i < DEG+1; ++i) {
    P[i] = var_create(i+1);
  }
}

var_t reimann_integral(var_t P[DEG+1]) {
  var_t loss = var_create(0);

  float step_size = (END-START)/N;
  for (size_t j = 0; j < N; ++j) {
    float x = START + j*step_size;
    var_t delta = poly_eval(P, x) - var_create(f(x));
    loss = loss + (delta*delta) * var_create(step_size);
  }

  return loss;
}

int main() {
  size_t runs = 10;
  float start_time, end_time;

  start_time = (float) clock() / CLOCKS_PER_SEC;
  for (size_t i = 0; i < 10; ++i) {
    var_t P[DEG+1];
    tape_t tape = tape_create(64);
    tape_load(tape);
    poly_init(P);
    var_t loss = reimann_integral(P);
    tape_destroy(tape);
  }
  end_time = (float) clock() / CLOCKS_PER_SEC;

  /* print average runtime in milliseconds */
  printf("%f", (end_time - start_time) / runs * 1000);
  return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <math.h>
#include <time.h>

const int N = 1000;  /* number of terms in the reimann sum */
const float START = 0;  /* the start of the integration interval */
const float END = 2;  /* the end of the integration interval */
#ifndef DEG
#warning "DEG set to default value 4"
const int DEG = 4;  /* degree of the polynomial proximation */
#endif

/* the function to approximate */
float f(float x) {
  if (x == 0) return 0;
  return exp(-1 / (x*x));
}

float poly_eval(float P[DEG+1], float x) {
  float val = P[0];
  float X = x;
  for (size_t i = 1; i < DEG+1; i++) {
    val = val + P[i] * X;
    X *= x;
  }
  return val;
}

void poly_init(float P[DEG+1]) {
  for (size_t i = 0; i < DEG+1; ++i) {
    P[i] = i+1;
  }
}

float reimann_integral(float P[DEG+1]) {
  float loss = 0;

  float step_size = (END-START)/N;
  for (size_t j = 0; j < N; ++j) {
    float x = START + j*step_size;
    float delta = poly_eval(P, x) - f(x);
    loss = loss + (delta*delta) * step_size;
  }

  return loss;
}

int main() {
  size_t runs = 10;
  float start_time, end_time;

  start_time = (float) clock() / CLOCKS_PER_SEC;
  for (size_t i = 0; i < runs; ++i) {
    float P[DEG+1];
    poly_init(P);
    float loss = reimann_integral(P);
  }
  end_time = (float) clock() / CLOCKS_PER_SEC;

  /* print average runtime in milliseconds */
  printf("%f", (end_time - start_time) / runs * 1000);
  return 0;
}
```

```
all: build

CC := clang
CFLAGS := -std=c++11 -O2 -lm

primal: primal.cpp
	$(if $(DEG),,$(error Must set DEG))
	$(CC) $(CFLAGS) -DDEG=$(DEG) primal.cpp -o primal_build_$(DEG)

reverse: reverse.cpp
	$(if $(DEG),,$(error Must set DEG))
	$(CC) $(CFLAGS) -DDEG=$(DEG) reverse.cpp -o reverse_build_$(DEG)

forward: forward.cpp
	$(if $(DEG),,$(error Must set DEG))
# I renamed GRADLEN to GL to avoid the overriding of GRADLEN
	$(eval GL := $(shell echo ${DEG}+1 | bc))
	$(CC) $(CFLAGS) -DDEG=$(DEG) -DGRADLEN=$(GL) forward.cpp -o forward_build_$(DEG)

forward_novec: forward.cpp
	$(if $(DEG),,$(error Must set DEG))
# I renamed GRADLEN to GL to avoid the overriding of GRADLEN
	$(eval GL := $(shell echo ${DEG}+1 | bc))
	$(CC) $(CFLAGS) -fno-vectorize -fno-slp-vectorize -DDEG=$(DEG) -DGRADLEN=$(GL) forward.cpp -o
forward_build_novec_$(DEG)

forward_gradlen: forward.cpp
	$(if $(DEG),,$(error Must set DEG))
	$(if $(GRADLEN),,$(error Must set GRADLEN))
	$(CC) $(CFLAGS) -DDEG=$(DEG) -DGRADLEN=$(GRADLEN) forward.cpp -o forward_build_gradlen_$(DEG)_$
(GRADLEN)

parallel: forward_parallel.cpp
	$(if $(DEG),,$(error Must set DEG))
	$(CC) $(CFLAGS) -DDEG=$(DEG) forward_parallel.cpp -o parallel_build_$(DEG)

parallel_workers: forward_parallel.cpp
	$(if $(DEG),,$(error Must set DEG))
	$(if $(WORKERS),,$(error Must set WORKERS))
	$(CC) $(CFLAGS) -DDEG=$(DEG) -DRI_WORKERS=$(WORKERS) forward_parallel.cpp -o
parallel_build_workers_$(DEG)_$(WORKERS)


# use -j option to run build in parallel
build: reverse forward forward_novec forward_gradlen parallel

clean:
	rm primal_build_* forward_build_* reverse_build_* parallel_build_*
```

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <math.h>
#include <time.h>

const int N = 1000;  /* number of terms in the reimann sum */
const float START = 0;  /* the start of the integration interval */
const float END = 2;  /* the end of the integration interval */
#ifndef DEG
#warning "DEG set to default value 4"
const int DEG = 4;  /* degree of the polynomial proximation */
#endif

#ifndef GRADLEN
#warning "GRADLEN set to default value 8"
#define GRADLEN 8
#endif
#include "../../forward.h"

/* the function to approximate */
float f(float x) {
  if (x == 0) return 0;
  return exp(-1 / (x*x));
}

var_t poly_eval(var_t P[DEG+1], float x) {
  var_t val = P[0];
  float X = x;
  for (size_t i = 1; i < DEG+1; i++) {
    val += P[i] * X;
    X *= x;
  }
  return val;
}

void poly_init(var_t P[DEG+1], size_t grad_start, size_t grad_end) {
  for (size_t i = 0; i < DEG+1; ++i) {
    P[i] = {.grad = {0}, .value = (float) i+1};
    if (i >= grad_start && i < grad_end) {
      P[i].grad[i - grad_start] = 1;
    }
  }
}

var_t reimann_integral(var_t P[DEG+1]) {
  var_t loss = {0};

  float step_size = (END-START) / N;
  for (size_t j = 0; j < N; ++j) {
    float x = START + j*step_size;
    var_t delta = poly_eval(P, x) - f(x);
    loss = loss + (delta*delta) * step_size;
  }

  return loss;
}

int main() {
  size_t runs = 10;
  float start_time, end_time;

  start_time = (float) clock() / CLOCKS_PER_SEC;
  for (size_t i = 0; i < runs; ++i) {
    for (size_t grad_start = 0; grad_start < DEG+1; grad_start += GRADLEN) {
      var_t P[DEG+1];
      poly_init(P, grad_start, grad_start + GRADLEN);
      var_t loss = reimann_integral(P);
    }
  }
  end_time = (float) clock() / CLOCKS_PER_SEC;

  /* print average runtime in milliseconds */
  printf("%f", (end_time - start_time) / runs * 1000);
  return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <math.h>
#include <time.h>
#include <pthread.h>
#include <assert.h>

const int N = 1000;  /* number of terms in the reimann sum */
const float START = 0;  /* the start of the integration interval */
const float END = 2;  /* the end of the integration interval */
const float ITERATIONS = 5000;  /* number of gradient descent iterations */
const float ALPHA = 0.001;  /* gradient descent speed */
#ifndef DEG
#warning "DEG set to default value 4"
const int DEG = 4;  /* degree of the polynomial proximation */
#endif

#ifndef GRADLEN
#define GRADLEN 64
#endif
#include "../../forward.h"

/* the function to approximate */
float f(float x) {
  if (x == 0) return 0;
  return exp(-1 / (x*x));
}

var_t poly_eval(var_t P[DEG+1], float x) {
  var_t val = P[0];
  float X = x;
  for (size_t i = 1; i < DEG+1; i++) {
    val += P[i] * X;
    X *= x;
  }
  return val;
}

void poly_print(float P[DEG+1]) {
  printf("polynomial: ");
  for (size_t i = 0; i < DEG; ++i) {
    printf("%f, ", P[i]);
  }
  printf("%f\n", P[DEG]);
}

#ifndef RI_WORKERS
#define RI_WORKERS 2
#endif
const size_t RI_CHUNKS = ((DEG+1 + GRADLEN-1) / GRADLEN);

typedef struct {
  size_t start_chunk;
  size_t end_chunk;
  float *P;
  float *grad;
  float value;
} ri_worker_param_t;

void *ri_worker(void *param_ptr) {
  ri_worker_param_t *param = (ri_worker_param_t *) param_ptr;

  for (size_t chunk_id = param->start_chunk; chunk_id < param->end_chunk; ++chunk_id) {
    var_t loss = {0};
    var_t P[DEG+1] = {0};
    for (size_t i = 0; i < DEG+1; ++i) {
      P[i].value = param->P[i];
      if (i >= chunk_id * GRADLEN && i < chunk_id * GRADLEN + GRADLEN) {
        P[i].grad[i - chunk_id * GRADLEN] = 1;
      }
    }

    float step_size = (END-START) / N;
    for (size_t j = 0; j < N; ++j) {
      float x = START + j*step_size;
      var_t delta = poly_eval(P, x) - f(x);
      loss = loss + (delta*delta) * step_size;
      if (loss.value != loss.value) {

      }
```

```
    }

    if (chunk_id == param->start_chunk) {
      param->value = loss.value;
    } else {
      /* assert(param->value == loss.value); */
    }
    for (size_t i = 0; i < GRADLEN && chunk_id * GRADLEN + i < DEG+1; ++i) {
      param->grad[chunk_id * GRADLEN + i] = loss.grad[i];
    }
  }

  return NULL;
}

float reimann_integral(float P[DEG+1], float grad[DEG+1]) {
  pthread_t worker_threads[RI_WORKERS];
  ri_worker_param_t worker_params[RI_WORKERS];

  assert(RI_CHUNKS > 0);
  int handled_chunks = 0;
  for (size_t worker_id = 0; worker_id < RI_WORKERS; ++worker_id) {
    size_t start_chunk = (size_t) ((float) RI_CHUNKS / RI_WORKERS * worker_id);
    size_t end_chunk = (size_t) ((float) RI_CHUNKS / RI_WORKERS * (worker_id+1));
    worker_params[worker_id] = {
      .start_chunk = start_chunk,
      .end_chunk = end_chunk,
      .P = P,
      .grad = grad,
    };
    int err = pthread_create(&worker_threads[worker_id], NULL, &ri_worker, &worker_params[worker_id]);
    if (err) {
      printf("pthread_create error %d", err);
      exit(1);
      return 0;
    }
    handled_chunks += end_chunk - start_chunk;
  }
  assert(handled_chunks == RI_CHUNKS);

  for (size_t worker_id = 0; worker_id < RI_WORKERS; ++worker_id) {
    int err = pthread_join(worker_threads[worker_id], NULL);
    if (err) {
      printf("pthread_join error %d", err);
      exit(1);
      return err;
    }
  }

  float value;
  for (size_t worker_id = 1; worker_id < RI_WORKERS; ++worker_id) {
    if (worker_params[worker_id].end_chunk - worker_params[worker_id].start_chunk > 0) {
      value = worker_params[worker_id].value;
    }
  }
  return value;
}

int main() {
  size_t runs = 10;
  float start_time, end_time;

  start_time = (float) clock() / CLOCKS_PER_SEC;
  for (size_t i = 0; i < runs; ++i) {
    float P[DEG+1];
    float loss_grad[DEG+1];
    float loss = reimann_integral(P, loss_grad);
  }
  end_time = (float) clock() / CLOCKS_PER_SEC;

  /* print average runtime in milliseconds */
  printf("%f", (end_time - start_time) / runs * 1000);
  return 0;
}
```

```
#!/bin/bash

gradlen=64

bench() {
  reverse=$(./reverse_build_"$1")
  forward=$(./forward_build_"$1")
  forward_novec=$(./forward_build_novec_"$1")
  forward_gradlen=$(./forward_build_gradlen_"$1"_"$gradlen")
  echo "$1","$reverse","$forward","$forward_novec","$forward_gradlen"
}

deg=(1 2 $(seq 4 4 512))
for d in ${deg[@]}; do
  make -j build DEG=$d GRADLEN=$gradlen > /dev/null &
done
wait

for d in ${deg[@]}; do
  bench $d
done

make clean > /dev/null
```

```
#!/bin/bash

d=500

bench() {
  parallel=$(./parallel_build_workers_"$d"_"$1")
  echo "$1","$parallel"
}

workers=$(seq 1 1 12)
for w in ${workers[@]}; do
  make -j parallel_workers DEG=$d WORKERS=$w > /dev/null &
done
wait

for w in ${workers[@]}; do
  bench $w
done

make clean
```

```
#!/bin/bash

bench() {
  reverse=$(./reverse_build_"$1")
  echo "$d","$reverse"
}

deg=(4 8 $(seq 4 16 512))
for d in ${deg[@]}; do
  make -j reverse DEG=$d > /dev/null &
done
wait

for d in ${deg[@]}; do
  bench $d
done

make clean
```

```
#!/bin/bash

bench() {
  parallel=$(./parallel_build_"$1")
  echo "$d","$parallel"
}

deg=(4 8 $(seq 4 16 512))
for d in ${deg[@]}; do
  make -j parallel DEG=$d > /dev/null &
done
wait

for d in ${deg[@]}; do
  bench $d
done

make clean
```

```
#!/bin/bash

deg=300

bench() {
  forward_gradlen=$(./forward_build_gradlen_"$deg"_"$1")
  echo "$1","$forward_gradlen"
}

gradlen=($(seq 4 1 512))
for gl in ${gradlen[@]}; do
  make -j forward_gradlen DEG=$deg GRADLEN=$gl > /dev/null &
done
wait

for gl in ${gradlen[@]}; do
  bench $gl
done

make clean
```