Algorithmic Differentiation in Computational Science, Engineering, and Finance

Introductory Case Studies

Edited by Uwe Naumann

LuFG Informatik 12:
Software and Tools for Computational Engineering
Department of Computer Science
RWTH Aachen University
D-52056 Aachen, Germany

www: `http://www.stce.rwth-aachen.de`
email: `naumann@stce.rwth-aachen.de`

Contents

Chapter 1

Sample Chapter

Uwe Naumann and Johannes Lotz

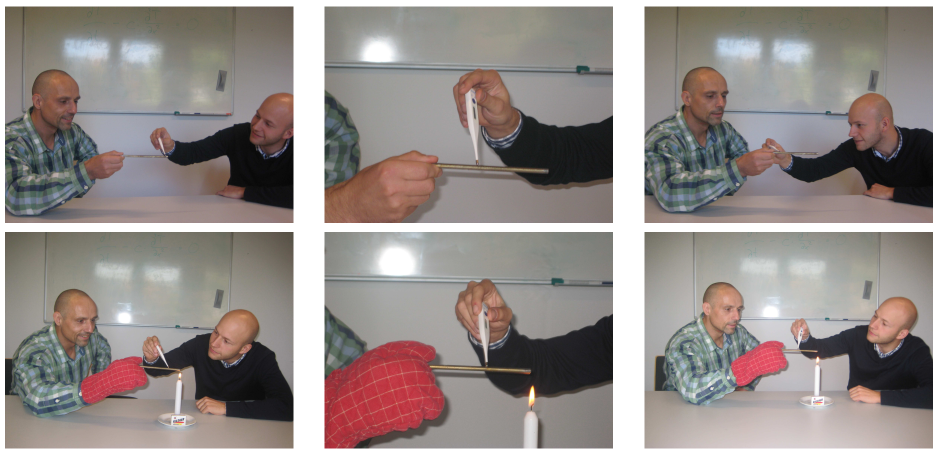

Figure 1.1. *Illustration of the Real-World Problem by the Authors: Naumann (Left) and Lotz (Right)*

TODO *Short intro and description of the real-world problem*

We consider the estimation of the *thermal diffusivity* $c \in I\!R$ of a given very thin stick. Thermal diffusivity describes the speed at which heat "spreads" within the material. High thermal diffusivity implies quick heat conduction. Our mathematical model depends on the unknown/uncertain parameter c, which is to be calibrated using experimentally obtained real-world measurements. The results of a numerical simulation of the temperature distribution within the stick after heating one of its two ends for some time are compared with given measurements. Refer to Figure 1 for a graphical illustration of the problem. The pictures in the upper row show the measuring of the initial temperature distribution in the stick. This process is continued while heating one end of the stick as shown in the pictures in the lower

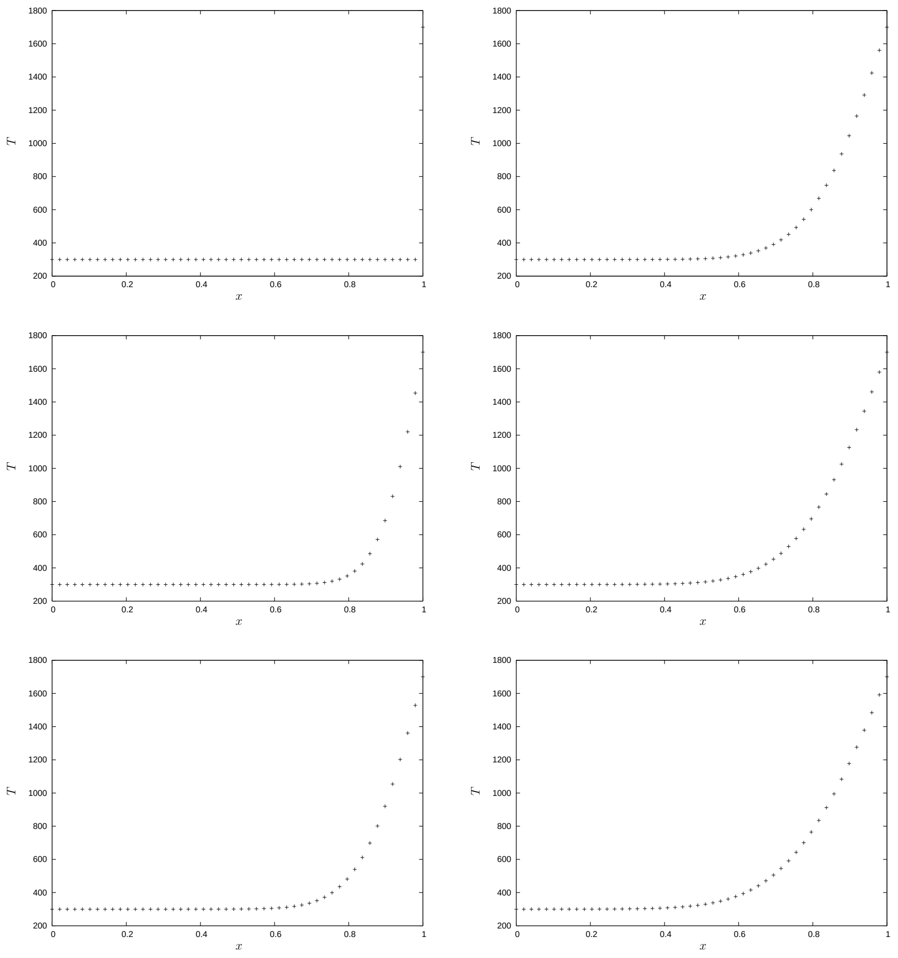

Figure 1.2. *Simulated Temperature Distribution for* $t = 0, 0.2, 0.4$ *(Left Column) and* $t = 0.6, 0.8, 1$ *(Right Column)*

row and where the temperature at the other end is kept constant.

1.1 Problem Description

TODO *description of the mathematical model*

The distribution of heat within the stick over time is modeled by a parabolic partial differential equation (PDE). We look for $T = T(t, x, c) : I\!R \times I\!R \times I\!R \to I\!R$ as the solution of an initial and boundary value problem for the 1D heat equation

$$\frac{\partial T}{\partial t} = c \cdot \frac{\partial^2 T}{\partial x^2} \tag{1.1}$$

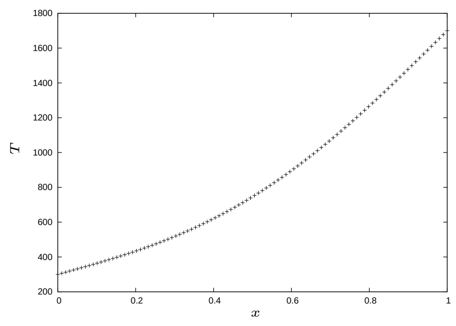

Figure 1.3. *Simulated Temperature Distribution at Time* $t = 5$

over the bounded domain (interval) $\Omega = (0, 1)$ with initial values $T(t = 0) = i(x)$ for $x \in \Omega$ and boundary values $T(x = 0) = b_0$ and $T(x = 1) = b_1$. Time is denoted by $t \in I\!R$. The position with the stick is represented by $x \in I\!R$. Figure 2 shows the development of the temperature distribution within the stick for $T(t = 0) = T(x = 0) = 300K$, $T(x = 1) = 1700K$,[1] and $c = 0.01$. While the effect of the flame on the temperature of various sections of the stick is not dramatic at the final time $t = 1$ a longer lasting exposure will eventually yield an increase of the temperature in the neighborhood of the left end of the stick beyond the comfort level. This effect is illustrated by Figure 3 showing the temperature distribution at time $t = 5$. Hence the protective pot cloth in Figure 1 ... Linearity of temperature T as a function of the position x within the stick lets the plot of the temperature distribution converge for $t \to \infty$ to a straight line that connects the two points (0,300) and (1,1700). The dependence on c vanishes asymptotically. Hence, we consider the calibration of the parameter c at time $t = 1$.

For the given value of c and given observations $O(x)$ at time $t = 1$ we aim to solve the unconstrained least squares problem

$$\min_{c \in I\!R} \int_\Omega (T(1, x, c) - O(x))^2 \, dx. \tag{1.2}$$

The observations are obtained through the measurement procedure illustrated in the lower row of pictures in Figure 1. We aim to estimate the unknown/uncertain material property c of the stick such that the real-world measurements are reproduced as well as possible by the implementation of the mathematical model.

1.2 Numerical Method

TODO *algorithmic description of primal solution method*

The continuous mathematical model needs to be translated into a discrete formulation in order to be able to solve it on today's computers. We use *finite*

[1]approximate temperature of a candle flame

difference quotients to replace both the spatial (see Section 0.2.1) and the temporal (see Section 0.2.2) differential terms in Equation (1). The integral in Equation (2) is discretized using a simple *quadrature rule* (see Section 0.2.3). See, for example, [4] for a gentle introduction to finite difference discretization methods.

1.2.1 Spatial Discretization

The given stick of unit length $[0, 1]$ is divided into $n-1$ sub-intervals of equal length

$$\Delta x = \frac{1}{n-1} \tag{1.3}$$

yielding a spatial discretization with $n-2$ inner points $x_1, \ldots, x_{n-2}$ in addition to the two end (boundary) points $x_0 = 0$ and $x_{n-1} = 0$. See Figure 4 for illustration.

Second spatial derivatives are approximated by second-order finite differences based on central finite difference approximation of the first derivatives at the center points

$$a_j \equiv \frac{x_{j-1} + x_j}{2} \tag{1.4}$$

and

$$b_j \equiv \frac{x_j + x_{j+1}}{2} \tag{1.5}$$

of the corresponding intervals. Figure 4 illustrates the approximation of $\frac{\partial^2 T^k}{\partial x^2}$ at grid point x_3 for $n = 6$ and, hence, $\Delta x = 0.2$. The first derivatives of T^k at a_3 and b_3 are approximated by backward finite differences. From

$$\frac{\partial T}{\partial x}(t, a_j, c) \approx \frac{T(t, x_j, c) - T(t, x_{j-1}, c)}{\Delta x} = \frac{T_j - T_{j-1}}{\Delta x} \tag{1.6}$$

for $j = 1, \ldots, n-2$ and

$$\frac{\partial T}{\partial x}(t, b_j, c) \approx \frac{T(t, x_{j+1}, c) - T(t, x_j, c)}{\Delta x} = \frac{T_{j+1} - T_j}{\Delta x} \tag{1.7}$$

for $j = 1, \ldots, n-2$ follows

$$\frac{\partial^2 T}{\partial x^2}(t, x_j, c) \approx \frac{\frac{\partial T}{\partial x}, c(t, b_j) - \frac{\partial T}{\partial x}(t, a_j, c)}{\Delta x} = \frac{T_{j+1} - 2 \cdot T_j + T_{j-1}}{(\Delta x)^2} \tag{1.8}$$

for $j = 1, \ldots, n-2$ and hence the system of ordinary differential equations (ODE)

$$\frac{\partial T_j}{\partial t} = r_j(c, \Delta x, T) \ , \quad j = 0, \ldots, n-1 \ , \tag{1.9}$$

where

$$r_j = 0 \qquad j \in \{0, n-1\} \tag{1.10}$$

$$r_j = \frac{c}{(\Delta x)^2} \cdot (T_{j+1} - 2 \cdot T_j + T_{j-1}) \qquad j \in \{1, \ldots, n-2\}, \tag{1.11}$$

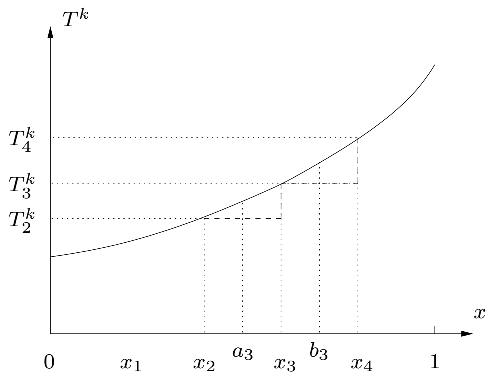

Figure 1.4. *Spatial Discretization: Central Finite Differences*

and where r denotes the right-hand side (also: residual) of the ODE resulting from the discretization of the second spatial derivative in Equation (1). The residual vanishes at both end points due to constant Dirichlet-type boundary conditions.

The ODE in Equation (9) is linear in T. Hence, it can be expanded into a first-order Taylor series at point $T \equiv 0$ ($T_j = 0$ for $j = 1, \ldots, n-2$) as follows:

$$\frac{\partial T}{\partial t} = \underbrace{r(c, \Delta x, 0)}_{=0} + \frac{\partial r}{\partial T}(c, \Delta x) \cdot (T - 0). \tag{1.12}$$

The residual at $T \equiv 0$ vanishes identically as a result of Equations (10) and (11). Linearity of the residual in T implies the independence of its first derivative from T, that is, the Jacobian $\frac{\partial r}{\partial T} = \frac{\partial r}{\partial T}(c, \Delta x)$ is constant with respect to temperature (and time). For given c and Δx a directional derivative in direction $T = (T_j)_{j=0,\ldots,n-1}$ is computed by the second term.

Example For $n = 6$ the Jacobian of the residual becomes

$$\frac{\partial r}{\partial T}(c, \Delta x) = \frac{c}{(\Delta x)^2} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{1.13}$$

The first and the last rows are identically equal to zero due to the constant Dirichlet boundary conditions (see Equation (10)).

The residual $r(c, \Delta x, T)$ is evaluated by the function

```
float[n] residual(float c, int n, float[n] T) {
  float[n] r:=0
  for i=1 to n-2
    r[j]:=c*(n-1)*(n-1)*(T[j+1] - 2*T[j] + T[j-1])
```

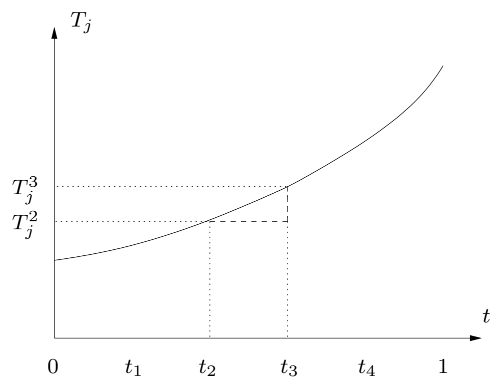

Figure 1.5. *Temporal Discretization: Backward Finite Difference*

```
  return r
}
```

and using Equation (3).

1.2.2 Temporal Discretization

The differential left hand side of the ODE in Equation (9) is approximated by a backward finite difference approximation yielding the *backward Euler method* (see, for example, [4]. Similar to Section 0.2.1, the unit time interval $[0,1]$ is decomposed into m sub-intervals of equal length

$$\Delta t = \frac{1}{m} \tag{1.14}$$

yielding a temporal discretization with m states $T_j^1, \ldots, T_j^m$, and a given initial state T_j^0 for $j = 0, \ldots, n-1$. Figure 5 illustrates the approximation of $\frac{\partial T_j}{\partial t}$ at time t_3 for $m = 5$ and, hence, $\Delta t = 0.2$. From

$$\frac{\partial T_j}{\partial t}(t_{k+1}, x, c) \approx \frac{T_j^{k+1} - T_j^k}{\Delta t}, \tag{1.15}$$

follows

$$\frac{T^{k+1} - T^k}{\Delta t} = r(c, \Delta x, T^{k+1}) = \frac{\partial r}{\partial T}(c, \Delta x) \cdot T^{k+1} \tag{1.16}$$

yielding the recurrence

$$\left(\Delta t \cdot \frac{\partial r}{\partial T}(c, \Delta x) - I\right) \cdot T^{k+1} = -T^k \tag{1.17}$$

with a constant system matrix on the left-hand side for given c, Δx, and Δt and where I denotes the identity in $I\!R^n$.

Methods for computing the Jacobian $\frac{\partial r}{\partial T}(c, \Delta x)$ of the residual by the function

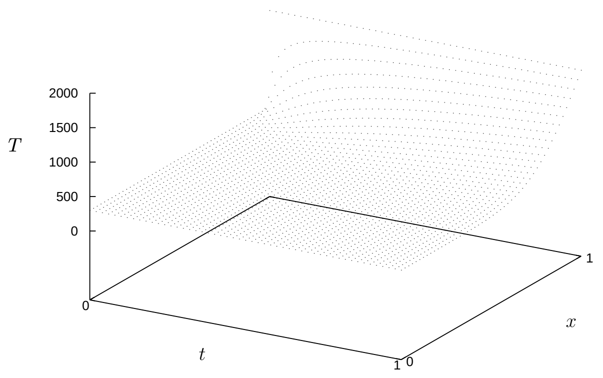

Figure 1.6. *Discrete Simulation of Temperature T as a Function of Time t and Position x.*

```
float[n][n] jacobian_residual(float c, int n, float[n] T)
```

are discussed in Section 0.3. Moreover, we require a LU decomposition

```
float[n][n] lu_decomp(float[n][n] A)
```

and a linear solver for a given LU decomposition of A

```
float[n] solve(float[n][n] LU, float[n] rhs)
```

for the solution of the system of linear equations

$$\underbrace{(\Delta t \cdot \frac{\partial r}{\partial T}(c, \Delta x) - I)}_{=:\mathrm{A}} \cdot T^{k+1} = \underbrace{-T^k}_{:=\mathrm{b}} \tag{1.18}$$

to be called during the simulation of the heat distribution by the following function:

```
float[n] sim(float c, int n, int m, float[n] T) {
  float[n][n] I // identity
  A := jacobian_residual(c,n,T)/m - I
  LU := lu_decomp(n,A)
  for i=1 to m {
    b:=-T
    T:=solve(LU,b)
  }
  return T
}
```

See Figure 6 for a spherical plot of the simulated function $T = T(x, t)$.

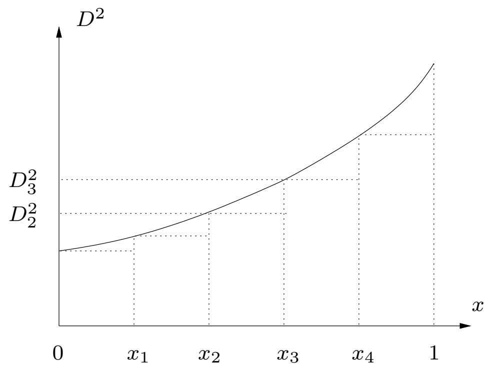

Figure 1.7. *Discretization of Objective: Quadrature*

1.2.3 Discretization of the Objective and Optimization

Discretization of the least-squares objective in Equation (2) using a simple quadrature rule over the given spatial discretization yields the objective function

$$y = f(c, n, m, T, O) = \frac{1}{n-1} \cdot \sum_{j=0}^{n-2} (T_j^m(c) - O_j)^2. \tag{1.19}$$

Figure 7 illustrates the quadrature formula. For example, the contribution due to the space interval $[x_2, x_3]$ is equal to

$$\Delta x \cdot D_2^2 = \frac{D_2^2}{n-1}$$

where $D_j = T_j^m(c) - O_j$.

The evaluation of the objective as a function of the initial temperature distribution includes the simulation of the temperature distribution at the target time. It can be implemented as follows:

```
float f(float c, int n, int m, float[n] T, float[n] O) {
  T:=sim(c,n,m,T)
  float y:=0
  for j=0 to n−2
    y:=y+(T[j]−O[j])*(T[j]−O[j])
  return y/(n−1)
}
```

The objective function f is optimized by a simple steepest gradient descent method with embedded local line search. Termination is defined by $\|\nabla f\| \leq \epsilon$ for $\epsilon \ll 1$ and it is based on the necessary first-order optimality condition $\nabla f = 0$. At each iteration i we take a step into negative gradient direction the size (α) of which is defined by recursive bisection such that a decrease in the value of the objective is ensured. This very basic approach turns out to be sufficient for the given simple problem. Formally, the steepest gradient descent method is described by the iteration

$$c_{i+1} = c_i - \alpha \cdot \nabla f(c_i) \quad \text{while } \|\nabla f\| > \epsilon.$$

1.3 Algorithmic Differentiation

TODO *Role of AD in the context of the numerical method*

Figure 8 visualizes the structure of the given simulation as a linearized DAG (refer to Chapter **??** for details). The local partial derivatives to be combined in forward or reverse order in tangent-linear or adjoint mode AD, respectively, are given in square brackets as edge labels. Linearity of the residual in T yields the independence of its Jacobian of T (node 6). The corresponding matrix $A = \Delta t \cdot \frac{\partial r}{\partial T}(c, \Delta x) - I$ (nodes 6,7,8) is decomposed into lower and upper triangular factors L and U (node 9) that enter the subsequent backward Euler steps (nodes 10,11 and 12,13). The first two are shown in Figure 8. For given right-hand sides ($-T_0$ and $-T_1$) the next step is computed by simple forward and backward substitution based on the given LU decomposition of A.

1.3.1 Tangent-Linear Residual

The tangent-linear residual returns the product of the Jacobian of the residual with a given vector $T^{(1)} \hat{=}$ t1_T$\in I\!R^n$ without prior accumulation of the Jacobian itself. It computes a *matrix-free* directional derivative.

```
float[n] t1_residual(float c, int n, float[n] t1_T) {
  float[n] t1_r:=0
  for i=1 to n-2
    t1_r[j]:=c*(n-1)*(n-1)*(t1_T[j+1]-2*t1_T[j]+t1_T[j-1])
  return t1_r
}
```

1.3.2 Jacobian of the Residual

The Jacobian $\frac{\partial r}{\partial T} \hat{=}$ J=J [:][:] $\in I\!R^{n\times n}$ of the residual is computed column-wise by letting the input vector $T^{(1)} \hat{=}$ t1_T $\in I\!R^n$ range over the Cartesian basis vectors in $I\!R^n$.

```
float[n][n] jacobian_residual(float c, int n, float[n] t1_T)
    {
  float[n][n] J
  t1_T:=0
  for i=0 to n-1 {
    t1_T[i]:=1
    J[:][i]:=t1_residual(c,n,t1_T)
    t1_T[i]:=0
  }
```

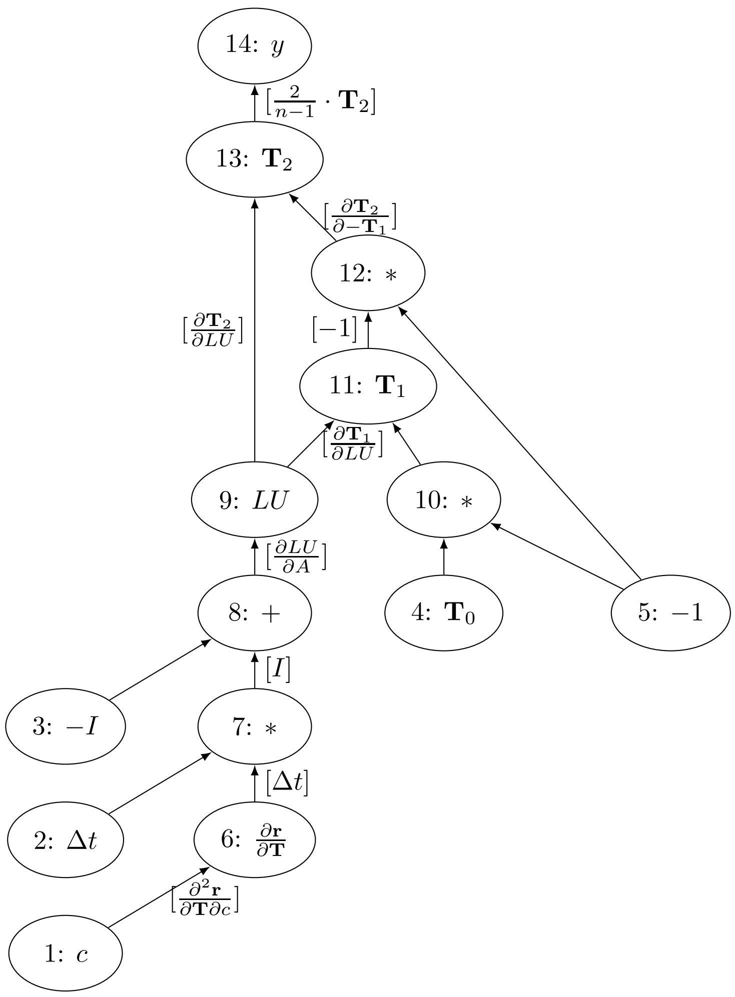
14: y
$[\frac{2}{n-1} \cdot \mathbf{T}_2]$
13: $\mathbf{T}_2$
$[\frac{\partial \mathbf{T}_2}{\partial -\mathbf{T}_1}]$
12: $*$
$[\frac{\partial \mathbf{T}_2}{\partial LU}]$
$[-1]$
11: $\mathbf{T}_1$
$[\frac{\partial \mathbf{T}_1}{\partial LU}]$
9: LU
10: $*$
$[\frac{\partial LU}{\partial A}]$
8: $+$
4: $\mathbf{T}_0$
5: -1
$[I]$
3: $-I$
7: $*$
$[\Delta t]$
2: Δt
6: $\frac{\partial \mathbf{r}}{\partial \mathbf{T}}$
$[\frac{\partial^2 \mathbf{r}}{\partial \mathbf{T} \partial c}]$
1: c

Figure 1.8. *Linearized DAG of Simulation Method*

```
  return J;
}
```

Sparsity of the (in this case tridiagonal) Jacobian can and should be exploited. Refer to [3] or [8] for details on corresponding compression techniques. Associated graph coloring problems are discussed in [1].

1.3.3 Gradient of the Objective

The gradient of the objective with respect to the free parameters that enter the simulation ($c \in I\!R$ in our case; refer to Section 0.5 for comments on the non-scalar case $c = c(x)$) is required by the steepest descent algorithm. It can be obtained by overloading the entire simulation and the following evaluation of the objective for either a tangent-linear or an adjoint AD type, for example, AD_TYPE $\in$ {dco_t1s_type, dco_a1s_type} when using `dco` (version 0.9; see Chapter **??** and [6]).

```
AD_TYPE f(AD_TYPE c, int n, int m, AD_TYPE[n] T, float[n] O)
    {
  T:=sim(c,n,m,T)
  float y:=0
  for j=0 to n-2
    y:=y+(T[j]-O[j])*(T[j]-O[j])
  return y/(n-1)
}
```

Adjoint mode should be preferred for large numbers of parameters where the actual number depends on the performance of the AD implementation provided by the given AD tool. For version 0.9 of `dco` this number ranges between 3 and 30 depending on the structure and complexity of the target simulation. It can be increased by further constant factors due to the potential need for *checkpointing* in order to manage the adjoint *data flow reversal* within the available memory resources. See [3, 6] for details on checkpointing methods.

AD of f implies AD of the function sim and thus the differentiation of the direct linear solver called during the simulation. While naive AD of direct linear solvers by overloading produces correct results its computational cost which is of order $O(n^3)$ can be reduced to $O(n^2)$ through the exploitation of further mathematical insight as shown in [2].

AD by overloading of f in tangent-linear mode implies the evaluation of a tangent-linear model of the Jacobian computation (see Section 0.3.1). Implicitly, second-order derivatives are computed. The computation of

$$A = \Delta t \cdot \frac{\partial r}{\partial T}(c, \Delta x) - I$$

yields the tangent-linear projection

$$A^{(2)} =< \frac{\partial A}{\partial c}, c^{(2)} > + < \frac{\partial A}{\partial T}, T^{(2)} >= \Delta t \cdot < \frac{\partial^2 r}{\partial T \partial c}(c, \Delta x), c^{(2)} >,$$

where $\frac{\partial^2 r}{\partial T^2} = 0$ due to the linearity of r in T.

Similarly, overloading in adjoint mode yields a second-order adjoint projection of the Hessian of the residual as

$$\begin{pmatrix} T_{(2)} \\ c_{(2)} \end{pmatrix} =< A_{(2)}, \frac{\partial A}{\partial (T,c)} > \tag{1.20}$$

$$= \Delta t \cdot < A_{(2)}, \frac{\partial^2 r}{\partial T \partial (T,c)}(c, \Delta x) > \tag{1.21}$$

$$= \begin{pmatrix} 0 \\ < A_{(2)}, \frac{\partial^2 r}{\partial T \partial c}(c, \Delta x) > \end{pmatrix} . \tag{1.22}$$

Linearity of r in T yields $T_{(2)} = 0 \in I\!R^n$.

1.4 Example

TODO *Fully worked out "Hello World" example (incl. sample runs and numerical results)*

We use `dco` for the computation of the gradient that is required by the steepest descent algorithm. Scalar thermal diffusivity is considered for simplicity. Both tangent-linear and adjoint modes are implemented. The Jacobian of the residual with respect to the temperature distribution is computed by a tangent-linear version of the residual that is generated by the source transformation AD tool `dcc`; see [6].

A basic implementation of the optimization algorithm in C++ is shown in Listing 1. Declaration, allocation, and initialization of the temperature distribution in lines 18-26 are followed by reading of the observations from a file in lines 28-32. The gradient of the objective is computed in either tangent-linear or adjoint modes in lines 42-45. Both derivative_t1s and derivative_a1s return the value and the gradient. The simple local line search algorithm in lines 47-55 turns out to speed up the steepest descent algorithm.

Listing 1.1. *Optimization algorithm using tangent-linear or adjoint mode for the computation of the gradient of the objective with respect to the free parameter.*

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <assert.h>
4  #include <fstream>
5  using namespace std;
6
7  #include "gauss.hpp"
8  #include "residual.hpp"
9  #include "t1_residual.hpp"
10 #include "residual_jacobian.hpp"
11 #include "sim.hpp"
12 #include "f.hpp"
13 #include "derivative_t1s.hpp"
14 #include "derivative_a1s.hpp"
15
16 int main(int argc, char *argv[]) {
17
18   assert(argc==4);
19   int n=atoi(argv[1]); // spatial grid
20   int m=atoi(argv[2]); // temporal grid
21   double c = 0.01;     // thermal diffusivity
22
23   // temperature distribution
24   double *const T = new double[n];
25   for (int i=0;i<n-1;++i) T[i]=300.0;
26   T[n-1]=1700.0;
27
```

```
  // reading observations:
  ifstream myfile("observations.csv", ios::in);
  double *const O=new double[n];
  for (int i=0;i<n;++i) myfile >> O[i];
  myfile.close();

  // steepest descent algorithm
  double eps=1e-2;
  double grad_obj=eps;
  int it=0;
  double *const local_T=new double[n]; // for line search

  while (fabs(grad_obj)>=eps) {
    double obj;
    if (atoi(argv[3]))
      derivative_a1s(c,n,m,T,O,obj,grad_obj);
    else
      derivative_t1s(c,n,m,T,O,obj,grad_obj);

    // local line search
    double local_obj=obj, alpha=1.0, local_c;
    while (local_obj>=obj) {
      local_c=c-alpha*grad_obj;
      for (int i=0;i<n;++i) local_T[i]=T[i];
      local_obj=f(local_c,n,m,local_T,O);
      alpha/=2.;
    }
    c=local_c;

    cout << " Step " << ++it << ": "
         << "c=" << c << "; obj=" << obj << ", "
         << "grad_obj=" << fabs(grad_obj) << endl;
  }
  delete [] local_T; delete [] O; delete [] T;

  return 0;
}
```

The computation of the gradient of the objective in tangent-linear mode is shown in Listing 2. We use the tangent-linear AD data type dco_t1s_type consisting of scalar value and directional derivative (tangent) components v and t, respectively. The directional derivative of the scalar thermal diffusivity t1s_c is set to 1.0 in line 12 to enable the retrieval of the partial derivative of the objective t1s_obj with respect to t1s_c in line 15.

Listing 1.2. *Tangent-linear gradient computation using* `dco`.

```
1 #include "dco.hpp"
```

```
 2 using namespace dco;
 3
 4 void derivative_t1s(const double& c, int n, int m,
 5                     const double *const T,
 6                     const double *const O,
 7                     double &obj, double &grad_obj) {
 8   gt1s<double>::type t1s_c=c;
 9   gt1s<double>::type *const t1s_T=new gt1s<double>::type[n];
10   for (int i=0;i<n;++i) t1s_T[i]=T[i];
11
12   derivative(t1s_c)=1.0;
13   gt1s<double>::type t1s_obj=f(t1s_c,n,m,t1s_T,O);
14   obj=value(t1s_obj);
15   grad_obj=derivative(t1s_obj);
16
17   delete [] t1s_T;
18 }
```

Gradient accumulation in adjoint mode yields the slightly more elaborate Listing 3. A tape is generated by overloading for the adjoint AD data type dco_a1s_type and it is stored persistently for later propagation of the adjoints by interpretation. A new tape is recorded in line 18 for each gradient computation (line 12) followed by a single interpretation in line 25 after setting the adjoint of the objective within the tape to 1.0 in line 21. The link between the program variable a1s_obj and its associated tape entry is established via the virtual address (va) component in dco_a1s_type as described in [6]. The scaled (with 1.0) partial derivative of the objective with respect to thermal diffusivity is retrieved in line 23.

Listing 1.3. *Adjoint gradient computation using* `dco`.

```
 1 #include "dco.hpp"
 2 using namespace dco;
 3
 4 typedef ga1s<double> DCO_MODE;
 5 typedef DCO_MODE::type DCO_TYPE;
 6 typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
 7
 8 void derivative_a1s(const double& c, int n, int m,
 9                     const double *const T,
10                     const double *const O,
11                     double &obj, double &grad_obj) {
12   DCO_MODE::global_tape=DCO_TAPE_TYPE::create();
13   DCO_TYPE *a1s_T = new DCO_TYPE[n];
14   for (int i=0;i<n;++i) a1s_T[i] = T[i];
15
16   DCO_TYPE a1s_c = c;
17   DCO_MODE::global_tape->register_variable(a1s_c);
18   DCO_TYPE a1s_obj=f(a1s_c,n,m,a1s_T,O);
```

```
  DCO_MODE::global_tape->register_output_variable(a1s_obj);

  derivative(a1s_obj)=1;
  DCO_MODE::global_tape->interpret_adjoint();
  grad_obj=derivative(a1s_c);
  obj=value(a1s_obj);
  DCO_MODE::global_tape->interpret_adjoint();
  DCO_TAPE_TYPE::remove(DCO_MODE::global_tape);

  delete [] a1s_T;
}
```

Listing 4 shows the implementation of the residual as described in Section 0.2. The use of the C++ template mechanism makes this implementation generic in the sense that the residual can be evaluated for varying data TYPEs of the active floating-point variables, for example, through instantiation with **double**, dco_t1s_type , or dco_a1s_type.

Listing 1.4. *Implementation of the residual.*

```
template <typename TYPE>
inline void residual(const TYPE& c, int n, TYPE *const T) {
  TYPE * const T_new =new TYPE[n];
  for (int i=1;i<n-1;++i)
    T_new[i]=c*(n-1)*(n-1)*(T[i-1]-2*T[i]+T[i+1]);
  T[0]=0;
  for (int i=0;i<n;++i) T[i]=T_new[i];
  T[n-1]=0;
  delete [] T_new;
}
```

The tangent-linear version of the residual with respect to the temperature shown in 5 was generated by `dcc` and edited for readability. It turns out to be independent of the primal temperature evaluation due to the previously established linearity of the residual.

Listing 1.5. *Implementation of the tangent-linear residual.*

```
template <typename TYPE>
inline void t1_residual(const TYPE& c, int n,
                        TYPE *const t1_T) {
  TYPE *t1_T_new=new TYPE[n];
  for (int i=1;i<n-1;++i)
    t1_T_new[i]=c*(n-1)*(n-1)*
                (t1_T[i-1]-2*t1_T[i]+t1_T[i+1]);
  t1_T[0]=0;
  for (int i=0;i<n;++i) t1_T[i]=t1_T_new[i];
  t1_T[n-1]=0;
  delete [] t1_T_new;
}
```

The tangent-linear residual is used in residual_jacobian for the computation of the Jacobian as a sequence of directional derivatives in direction of the Cartesian basis vectors t1_T in $I\!R^n$. Sparsity is not exploited; see comments in Section 0.3.

Listing 1.6. *Implementation of the Jacobian computation using the tangent-linear residual.*

```
template <typename TYPE>
inline void residual_jacobian(const TYPE& c, int n,
                              TYPE *const J) {
  TYPE *t1_T = new TYPE[n];
  for (int i=0;i<n;++i) t1_T[i]=0;
  for (int i=0;i<n;++i) {
    t1_T[i]=1;
    t1_residual(c,n,t1_T);
    for (int j=0;j<n;++j) {
      J[j*n+i]=t1_T[j];
      t1_T[j]=0;
    }
  }
  delete [] t1_T;
}
```

Listing 7 implements the simulation of the temperature distribution over time as described in Section 0.2.2.

Listing 1.7. *Implementation of the simulation of temperature distribution.*

```
template <typename TYPE>
inline void sim(const TYPE& c, int n, int m, TYPE *const T)
    {

  TYPE * const A=new TYPE[n*n];

  residual_jacobian(c,n,A);

  for (int i=0;i<n;++i) {
    for (int j=0;j<n;++j)
      A[i*n+j]=A[i*n+j]/m;
    A[i+i*n]=A[i+i*n]-1;
  }

  lu_decomp(n,A);

  for (int j=0;j<m;++j) {
    for (int i=0;i<n;++i) T[i]=0-T[i];
    solve(n,A,T);
```

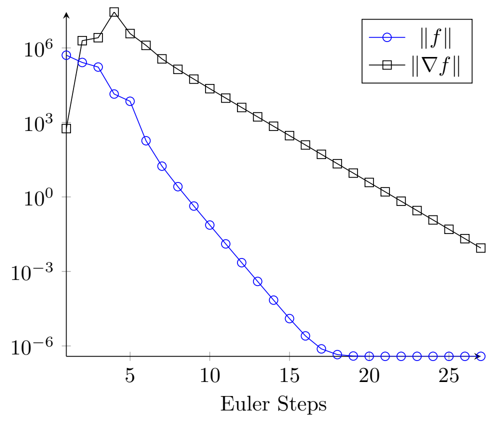

Figure 1.9. *Convergence Behavior for $n = 160$.*

```
    }

    delete [] A;
}
```

The implementation of the objective according to Section 0.2.3 is shown in Listing 8.

Listing 1.8. *Implementation of the objective function.*

```
template <typename TYPE>
inline TYPE f(const TYPE& c, int n, int m,
              TYPE *const T,
              const double *const O) {

  sim(c,n,m,T);

  TYPE f=0;
  for (int i=0;i<n-1;++i)
    f=f+(T[i]-O[i])*(T[i]-O[i]);
  f=f/(n-1);

  return f;

}
```

Figure 9 illustrates the convergence behavior of the steepest descent algorithm. Both the value of the objective and the norm of the gradient (first-order local optimality condition) decrease monotonously with the growing number of steepest descent steps.

n	10	20	50	100	150	200	250	300	350
Tape in mb	0.63	2.5	17.3	83.9	222.5	456.1	807.6	1299.9	—

Table 1.1. *Memory Consumption of Tape.*

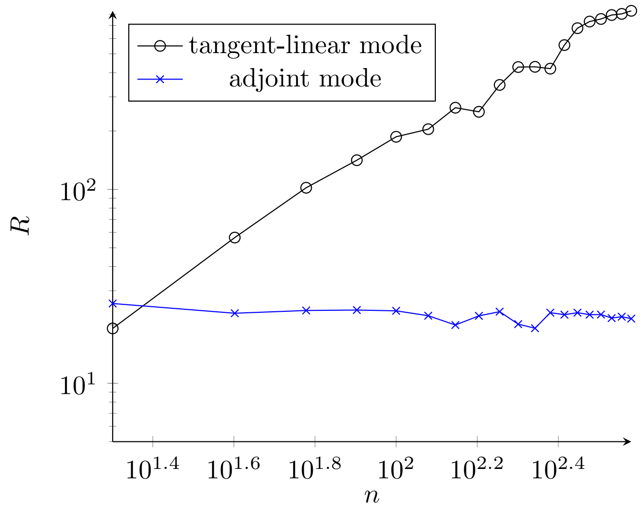

Figure 1.10. *Relative Run Time R for Growing Problem Sizes n.*

The memory requirement of adjoint mode AD grows linearly with computational cost of an evaluation of the objective. On our target architecture (2gb of RAM) The presented example runs out of memory for $n = 350$ as illustrated in Table 1.

1.5 Conclusion and Outlook

TODO *toward the real world*

The intention of this chapter was to give the reader a first impression on the use of AD in the context of a real-world application problem. Calibration of unknown / uncertain thermal diffusivity to given observations for the temperature distribution within a stick after some period of exposure to heating / cooling at both ends exhibits many properties of practically more relevant problems. We expect the careful reader of this chapter to be prepared for the potential transition to more complex AD scenarios, including non-scalar parameter calibration, second-order local optimization methods, approaches to the exploitation of sparsity of derivative tensors, and checkpointing techniques for large-scale first- and second-order adjoint simulations.

For further motivation consider Equation (1) with spatially varying thermal diffusivity $c(x)$ (after discretization c_i). The gradient of the objective with respect to the discrete $c = (c_i)_{i=0,\ldots,n-1}$ becomes a vector of size n. In tangent-linear mode it is computed as n inner products with the Cartesian basis vectors in $\mathbb{R}^n$. A single run of the adjoint code performs the same task at considerably lower computational cost for $n \gg 1$.. Figure 10 shows the relative computational cost

$$R = \frac{\text{run time for gradient computation}}{\text{run time for function evaluation}}$$

of the gradient computation in tangent-linear versus adjoint mode. The relative cost of gradient computation in tangent-linear mode grows linearly with n while in adjoint mode it remains constant. Availability of an adjoint simulation may turn out to be crucial for the applicability of gradient-based methods to large-scale problems in Computational Science, Engineering, and Finance.

In 3D (as opposed to the 1D scenario described in this chapter) the ratio of the computational cost of solving the linear system (with a complexity of $O(n^3)$) with respect to the overall computing time will grow considerably. In adjoint mode the size of the tape is likely to exceed the given memory bounds even for moderate problem sizes. Checkpointing will become a crucial ingredient of a robust and scalable adjoint simulation. Moreover, the application of AD to the (direct) linear solver can (and should) be avoided by treating the linear system as an intrinsic function as outlined in [2]. This approach reduces the memory requirement further from $O(n^3)$ to $O(n^2)$.

Algorithmic differentiation of non-trivial numerical simulation code is typically far from *automatic*. Educated users of robust and flexible AD tools are capable of generating highly sophisticated solutions for many practically relevant problems in Computational Science, Engineering, and Finance by targeted application of AD to relevant sections of the simulation. The availability of massively parallel high-performance computing platforms further complicates the semantic transformation of modern numerical simulations including AD. Nevertheless we have been observing a constantly growing interest in this mathematically and algorithmically exciting as well as practically relevant subject – and we hope to continue doing so.

Bibliography

[1] A. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.

[2] Mike B Giles. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. *Advances in Automatic Differentiation*, pages 35–44, 2008.

[3] A. Griewank and A. Walter. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation (2. Edition).* SIAM, Philadelphia, 2008.

[4] M. Heath. *Scientific Computing. An Introductory Survey.* McGraw-Hill, New York, 1998.

[5] U. Naumann. DAG reversal is NP-complete. *J. Discr. Alg.*, 7:402–410, 2009.

[6] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.* Number 24 in Software, Environments, and Tools. SIAM, 2012.

[7] U. Naumann and O. Schenk, editors. *Combinatorial Scientific Computing*, Computational Science Series. Chapman & Hall / CRC Press, Taylor and Francis Group, 2012.

[8] U. Naumann and A. Walther. Combinatorial problems in Algorithmic Differentiation. In *[7]*, pages 129–162, 2011.