Revolve - Program-Reversals for Time-stepping Procedures

Philipp Stumm* Andrea Walther†

Abstract The C/C++ package revolve creates checkpointing schedules for adaptive and non-adaptive time stepping procedures. For non-adaptive time steppings, this package provides binomial checkpointing schedules, if the states are only stored in main memory and multistage binomial checkpointing schedules, if the states are held in main memory and on disc too. For adaptive time stepping procedures, this package provides binomial-like checkpointing schedules that are optimal over a wide range. We provide three example programs for the use of revolve for an example adjoint ODE to be integrated. It is shown how all three checkpointing approaches are implemented into the adjoint process.

Keywords PDE constrained optimal control, adjoints, program reversals

1 Introduction

In time-dependent flow control as well as in the framework of goal oriented time-dependent a posteriori error control, the calculation of adjoint information forms a basic ingredient to generate the required derivatives for the cost functional (see e.g. [1, 4]). However, the corresponding computations may become extremely tedious if possible at all. This is mainly due to the sheer size of the resulting discretized problem as well as its nonlinear character which imposes a need to keep track of the complete forward solution in order to be able to integrate the corresponding adjoint differential equation backwards. This fact still forms quite often a main bottleneck in the overall optimization process despite the ever growing size of memory devices.
Checkpointing strategies use a small number of memory units (checkpoints) to store the system state at distinct times. The recomputation of required states that have not been stored is started from these checkpoints. Several checkpointing techniques have been developed. These methods seek for an acceptable compromise between memory requirement and runtime increase.
Depending on the time stepping procedure one distinguishes between two main

*Institut für Wissenschaftliches Rechnen, Technische Universität Dresden, Dresden, Germany, Philipp.Stumm@tu-dresden.de
†Institut für Mathematik, Universität Paderborn, Paderborn, Germany, Andrea.Walther@uni-paderborn.de

checkpointing strategies. If the number of time steps to be performed is not known a priori, one has to use online checkpointing strategies. This means that one has to decide on the fly where to place a checkpoint. The used online checkpointing approaches in revolve can be found in [6, 8]. The number of time steps to be performed for integrating the adjoint ODE is arbitrary because revolve switches to the corresponding algorithms automatically. If the number of time steps is known a priori, one can decide before the forward calculation where to place the checkpoints. In this case, one uses offline checkpointing strategies. Here, we consider only the binomial checkpointing strategy because it yields a provably optimal, i.e. minimal, amount of recomputations [2, 3]. If one extends this approach such that some checkpoints can be stored on disc too, one can reduce the number of additional time steps. In this case, one has to take the access cost to one checkpoint into account. The multistage offline binomial checkpointing approach yields checkpointing schedules where checkpoints are held in the main memory and on disc with time-minimal costs [5].
In the package revolve, there are three example programs how to use the checkpointing class Revolve for the three presented checkpointing approaches for the integration of time-dependent adjoint equations. The binomial checkpointing stategy is presented in revolve_binomial.cpp, the multistage technique in revolve_multistage.cpp, and the online approach in revolve_online.cpp. Additionally, we provide the example program no_checkpointing.cpp that integrates our adjoint equation without checkpionting. For all four programs, we consider the following ODE

$$\begin{aligned} \frac{dy_1}{dt} &= 0.5 * y_1(t) + u(t) \qquad y_1(0) = 1 \\ \frac{dy_2}{dt} &= y_1(t)^2 + 0.5 * u(t)^2 \quad y_2(0) = 0 \end{aligned} \tag{1}$$

to be integrated and an objective evaluated at $t = 1$ to be minimized

$$J(y) = y_2(1) \rightarrow \min!$$

Then the adjoint equations fulfill

$$\begin{aligned} \frac{dl_1}{dt} &= -0.5 * l_1(t) - 2 * y_1(t) * l_2(t) \quad l_1(1) = 0 \\ \frac{dl_2}{dt} &= 0 \qquad\qquad\qquad\qquad\qquad\qquad\quad l_2(1) = 1. \end{aligned} \tag{2}$$

The solution of this optimal control problem is given by (see [7])

$$\begin{aligned} y_1^*(t) &= (2 * e^{3t} + e^3)/(e^{3t/2} * (2 + e^3)) \\ y_2^*(t) &= (2 * e^{3t} - e^{6-3t} - 2 + e^6)/((2 + e^3)^2) \\ u^*(t) &= (2 * e^{3t} - e^3)/(e^{3t/2} * (2 + e^3)) \\ l_1^*(t) &= (2 * e^{3-t} - 2 * e^{2t})/(e^{t/2} * (2 + e^3)) \\ l_2^*(t) &= 1. \end{aligned}$$

First, we present the program no_checkpointing.cpp that computes the approximate solution of (1) at $t = 1$ and of (2) at $t = 0$ for $u = u^*$ without checkpointing, because this is our basic program to be extended by checkpointing routines. The time stepping is performed by a second order explicit Runge-Kutta method with constant step size h. We denote the number of time steps by n, hence $h = 1/n$. The computed solutions almost coincide with the optimal solutions depending on h. In order to solve the adjoints numerically, we divide the solution process into three parts.

1. Forward loop - Determination of the states $y_0, y_1, \ldots y_{n-1}$
2. Firstturn - Determination of state y_n and adjoint state l_{n-1}
3. Backward loop - Determination of the adjoint states $l_{n-2}, \ldots l_0$

During the forward loop, every state y_i has to be stored before the new state y_{i+1} is determined. The adjoint calculation starts with the firstturn. During the backward loop, every state y_i stored has to be available before the adjoint calculation. The entire integration process is given by

for $i = 0, \ldots, n-1, 1$
  store y_i, $y_{i+1} = F(y_i)$ // Forward loop
end for
$y_n = F(y_{n-1})$, $l_{n-1} = \bar{F}(y_{n-1}, l_n)$ // Firstturn
for $i = n-2, 0, -1$
  restore y_i, $l_i = \bar{F}(y_i, l_{i+1})$ // Backward loop
end for

Next, we consider for our example program no_checkpointing.cpp the interfaces for the forward and adjoint time integration. Note that these interfaces are adapted to the this special simple integration. In other applications, it may look different. All states determined in the forward-loop are stored into an array Y_C. The routines for the forward time step F, the adjoint time step $\bar{F}$, for storing a state in and retrieving states out of a checkpoint are given as follows

```
void advance(Y,Y_H,t,h)   // forward integration
double Y[2]               //  old state y(i)
double Y_H[2]           //  new state determined y(i+1)
double t                   // time t
double h                  // step length h

void adjoint(L_H,Y_H,L,t,h) // adjoint integration
double L_H[2]               // old adjoint state l(i+1)
double Y_H[2 ]             // state y(i)
double L[2]                // new adjoint state determined l(i)
double t                      // time t
```

```
double h                       // step length h

void store(Y_H,Y_C,t,i)  // store a state into the checkpoint i
double Y_H[2]                 // state to be stored
double **Y_C                  // checkpoint field
double t                         // time t
int i                               // index for storage

void restore(Y_H,Y_C,t,i) // retrieve a state out of the cp i
double Y_H[2]                // state to be retrieved
double **Y_C            // checkpoint field
double *t                    // time t
int i                         // index for retrieval
```

Then, the full solution process determining the states $y_0, \ldots, y_n$ and the adjoint states $l_0, \ldots, l_{n-1}$ without checkpointing as described in no_checkpointing.cpp is given by

```
// Initialization
h = 1./steps; t = 0.; Y[0] = 1.; Y[1] = 0.;
// forward loop
 for(int i=0;i<steps-1;i++)    {
          store(Y,Y_C,t,i);
          Y_H[0] = Y[0]; Y_H[1] = Y[1];
          advance(Y,Y_H,t,h);
          t += h;
}
// firstturn
advance(Y_final,Y_H,t,h);
L[0] = 0.; L[1] = 1.; t = 1.-h; L_H[0] = 0.; L_H[1] = 1.;
adjoint(L_H,Y_H,L,t,h);
// backward loop
for(int i=steps-2;i>=0;i--)  {
          restore(Y,Y_C,&t,i);
          L_H[0] = L[0]; L_H[1] = L[1];
          adjoint(L_H,Y,L,t,h);  t = t - h;     }
```

This paper is structured as follows. In the next section, we present the checkpointing class Revove the user always has to deal with. In the following three sections, we present the three example programs for the corresponding checkpointing approaches and describe how to employ the class Revolve into these programs.

2 The Checkpointing class Revolve

The user who wants to use a checkpointing approach has to deal with the class Revolve. For this reason, it is presented in detail in this section. The class Revolve with its most important functions looks

```
class Revolve
{
    Revolve(steps,snaps);   // offline checkpointing
    Revolve(steps,snaps,snaps_in_ram);//offline with diff. stores
    Revolve(snaps);            // online checkpointing

    ACTION::action revolve(); // tells the user what to do
    void turn(final);     // function for Online Checkpointing
    int getcheck();     // function for store or restore
    int getcheckram(); // function for (re)store for Multistage CP
    int getcheckrom(); // function for (re)store for Multistage CP
    int getcapo();      // function for advance
    int getinfo();       // function for error
    int getoldcapo();  // function for advance
    bool getwhere();  // function for Multistage CP
};
```

The main routine of this class is revolve(). The return value is of type

```
 enum action {
    advance,takeshot,restore,firsturn,youturn,terminate,error}.
```

The namespace ACTION is used because terminate also exists for the namespace std being used in revolve.h. The return value tells the user what to do as described as follows:

- takeshot - store a state into a checkpoint
- advance - make forward calculations
- firsturn - do the first reversal step
- youturn - do one reversal step
- restore - retrieve a state from a checkpoint
- error - an error occured

After the initialization, the integration process, hence all three integration steps - forward loop, firstturn and backward loop - has to be included into a do-while-loop where the function revolve is called each time. Therefore, the whole loop in every checkpointing approach reads

```
...   // Initializations to be done
do {
  whatodo = r->revolve();
  if (whatodo == ACTION::takeshot)  {  }
  if (whatodo == ACTION::advance)  { }
  if (whatodo == ACTION::firsturn)  { }
  if (whatodo == ACTION::youturn) { }
  if (whatodo == ACTION::restore)  { }
  if (whatodo == ACTION::error)
     cerr << " irregular termination of revolve " << endl;
}
while((whatodo!=ACTION::terminate)&&(whatodo!=ACTION::error));
```

The special actions to be performed in the if-clauses are explained later for every single checkpointing approach. If an error occurs, the program is immediately aborted and the user may ask for more detailed information by using the function getinfo(). The possible return values of this function with an explanation of the occured error is given as follows.

- 10 - number of checkpoints stored exceeds the predefined value checkup
- 11 - number of checkpoints stored exceeds the value snaps
- 12 - number of time steps must be increased
- 13 - number of checkpoints must be increased
- 14 - number of snaps exceeds the predefined value snapsup
- 15 - number of reps exceeds the predefined value repsup

The current values for checkup, snapsup, and repsup are

- checkup = 10000
- repsup = 6400
- snapsup = 10000

This means that no more than 10000 checkpoints can be used and the repetition number should be less than 6400. These values can be found in revolve.h and changed, if necessary.
At the beginning of the three checkpointing programs, the user may ask for additional information of the return values of the function revolve during the adjoint integration process by defining the variable info. This variable may have the following values

1. no information,
2. write only the takeshots,

3. write all actions the routine revolve returns.

The next sections are devoted to each of the three checkpointing approaches and to the three example programs, respectively. It is shown how the class Revolve is used and how the if-clauses of the do-while-loop are filled.

3 Offline Checkpointing - revolve_binomial.cpp

The constructor for binomial offline checkpointing is Revolve(steps,snaps), where steps denotes the number of time steps and snaps the number of checkpoints used. There are three functions of the class Revolve that are needed for binomial offline checkpointing. The return value of the function getcheck() tells the user in which checkpoint the current state is to be stored or the contents of which checkpoints should be restored. This function has to be used for the actions ACTION::takeshot and ACTION::restore. In the case of ACTION::advance, one has to perform some forward time steps from the return value of getoldcapo() to the return value of getcapo(). Then, the solution of the adjoint equations (1) and (2) fitting into the binomial offline checkpointing approach is given by

```
r=new Revolve(steps,snaps)
do {
   whatodo = r->revolve();
   if (whatodo == ACTION::takeshot)
       store(Y,Y_C,t,r->getcheck());
   if (whatodo == ACTION::advance) {
       for(int j=r->getoldcapo();j<r->getcapo();j++) {
           Y_H[0] = Y[0]; Y_H[1] = Y[1];
           advance(F,F_H,t,h); t += h;   } }
   if (whatodo == ACTION::firsturn) {
       advance(Y_final,Y,t,h);
       L[0] = 0.; L[1] = 1.; t = 1.-h; L_H[0] = 0.; L_H[1] = 1.;
       adjoint(L_H,Y_H,L,t,h); }
   if (whatodo == ACTION::youturn)  {
       L_H[0] = L[0]; L_H[1] = L[1];
       adjoint(L_H,Y,L,t,h); t = t - h; }
   if (whatodo == ACTION::restore)
       restore(Y,Y_C,&t,r->getcheck());
   if (whatodo == ACTION::error)
       cerr << " irregular termination of revolve \n";
}
while ((whatodo != ACTION::terminate)&&(whatodo != ACTION::error));
```

4 Online Checkpointing - revolve_online.cpp

The constructor for online checkpointing is Revolve(snaps), where snaps denotes the number of checkpoints used. First, the user has to store the initial state in checkpoint 0, hence

store(Y,Y_C,0.,0)

There are at least four functions of the class Revolve the user has to deal with for online checkpointing. The return value of the function getcheck() tells the user in which checkpoint a current state is to be stored or the content of which checkpoint has to be restored. This function has to be used for the actions ACTION::takeshot and ACTION::restore. If the action to be performed is ACTION::advance, one has to perform some forward time steps from the return value of getoldcapo() to the return value of getcapo(). Since the number of time steps is unknown a priori, the user has to tell Revolve when to start the reversal process. This is done by using the function turn(final) with the argument final telling Revolve the maximum number of time steps performed. The condition in our case for starting the reversal is capo = final-1 but it may be completely different in other applications. Then, the solution of the adjoint equations (1) and (2) fitting into the online checkpointing approach is given by

```
r=new Revolve(snaps);
store(Y,Y_C,0.,0);
do {
   whatodo = r->revolve();
   if (whatodo == ACTION::takeshot)
      store(Y,Y_C,t,r->getcheck());
   if (whatodo == ACTION::advance) {
      if (r->getcapo() >= final-1)
          r->turn(final);
      for(int j=r->getoldcapo();j<r->getcapo();j++)  {
         Y_H[0] = Y[0]; Y_H[1] = Y[1];
         advance(Y,Y_H,t,h); t += h; } }
   if (whatodo == ACTION::firsturn)  {
      advance(Y_final,Y,t,h);
      L[0] = 0.; L[1] = 1.; t = 1.-h; L_H[0] = 0.; L_H[1] = 1.;
      adjoint(L_H,Y_H,L,t,h); }
   if (whatodo == ACTION::youturn) {
      L_H[0] = L[0];  L_H[1] = L[1];
      adjoint(L_H,Y,L,t,h); t = t - h; }
   if (whatodo == ACTION::restore)
      restore(Y,Y_C,&t,r->getcheck());
   if (whatodo == ACTION::error)
      cerr << " irregular termination of revolve " << endl;
}
```

```
while((whatodo!=ACTION::terminate)&&(whatodo!=ACTION::error));
```

5 Multistage Checkpointing - revolve_multistage.cpp

Revolve(steps,snaps,snaps_in_ram) is the constructor for multistage offline checkpointing, where steps denotes the number of time steps, snaps the overall number of checkpoints and snaps_in_ram the number of checkpoints held in main memory. Hence, the number of checkpoints held on disc is snaps-snaps_in_ram. There are four functions of the class Revolve the user has to deal with for binomial multistage offline checkpointing. If the action to be performed is ACTION::advance, one has to perform some forward time steps from the return value of getoldcapo() to the return value of getcapo(). There also exists the function getwhere() telling the user to place the checkpoint either in main memory or on disc. If the return value of getwhere() is true, the state has to be stored or restored in main memory otherwise on disc. Since the storage places in the checkpointing distribution may not be equally distributed, there exists two checkpointing distributions - one for the states held in main memory and the other one for the states on disc. The return value of the function getcheckram() tells the user in which checkpoint in main memory to store or restore a state if getwhere() returns true. The return value of the function getcheckrom() tells the user in which checkpoint on disc to store or restore a state if getwhere() returns false. These three function have to be used for the actions ACTION::takeshot and ACTION::restore. Additionally, the two extra functions are provided for storing and restoring a state into or out of a checkpoint on disc. Here are the interfaces.

```
void store_rom(Y_H,t,check) // store a state on disc
double Y_H[2]              // state Y to be stored
double t                   // time t
int check                    // index of checkpoint

void restore_rom(Y_H,t,check)
double Y_H[2]          // state Y to be restored
double *t                 // time t
int check               // index of checkpoint
```

Note, that these interfaces may differ for other applications. It is only adapted to our example. Then, the solution of the adjoint equations (1) and (2) fitting into the binomial offline multistage checkpointing approach reads

```
r=new Revolve(steps,snaps,snaps_in_ram);
do {
    whatodo = r->revolve();
    if (whatodo == ACTION::takeshot) {
        if(r->getwhere())
            store(Y,Y_C,t,r->getcheckram());
        else
```

```
            store_rom(Y,t,r->getcheckrom());  }
    if (whatodo == ACTION::advance)   {
        for(int j=r->getoldcapo();j<r->getcapo();j++) {
            Y_H[0] = Y[0]; Y_H[1] = Y[1];
            advance(Y,Y_H,t,h); t += h; } }
    if (whatodo == ACTION::firsturn) {
        advance(Y_final,Y,t,h);
        L[0] = 0.; L[1] = 1.; t = 1.-h; L_H[0] = 0.; L_H[1] = 1.;
        adjoint(L_H,F_H,L,t,h); }
    if (whatodo == ACTION::youturn) {
       L_H[0] = L[0]; L_H[1] = L[1];
       adjoint(L_H,Y,L,t,h); t = t - h; }
    if (whatodo == ACTION::restore) {
        if(r->getwhere())
            restore(Y,Y_C,&t,r->getcheckram());
        else
            restore_rom(Y,&t,r->getcheckrom()); }
    if (whatodo == ACTION::error)
        cout << " irregular termination of revolve " << endl;
        abort(); }
}
while((whatodo!=ACTION::terminate)&&(whatodo!=ACTION::error));
```

Appendix

We list all functions used in the sections before where we presented the interfaces in the appendix.

```
void advance(double F[2],double F_H[2],double t,double h)
{
   double k0[2],k1[2],G[2];
   func(F_H,t,k0);
   G[0] = F_H[0] + h/2.*k0[0];
   G[1] = F_H[1] + h/2.*k0[1];
   func(G,t+h/2.,k1);
   F[0] = F_H[0] + h*k1[0];
   F[1] = F_H[1] + h*k1[1];
}

void adjoint(double L_H[2],double F_H[2],double L[2],double t,
             double h)
{
   double k0[2],k1[2],G[2],BH[2],Bk0[2],Bk1[2],BG[2];
   func(F_H,t,k0);
   G[0] = F_H[0] + h/2.*k0[0];
```

```
   G[1] = F_H[1] + h/2.*k0[1];
   func(G,t+h/2.,k1);
   L[0] = L_H[0];
   L[1] = L_H[1];
   Bk1[0] = h*L_H[0];
   Bk1[1] = h*L_H[1];
   func_adj(Bk1,G,BG);
   L[0] += BG[0];
   L[1] += BG[1];
   Bk0[0] = h/2.*BG[0];
   Bk0[1] = h/2.*BG[1];
   func_adj(Bk0,F_H,BH);
   L[0] += BH[0];
   L[1] += BH[1];
}

void store(double F_H[2], double **F_C,double t,int i)
{
   F_C[0][i] = F_H[0];
   F_C[1][i] = F_H[1];
   F_C[2][i] = t;
}

void restore(double F_H[2], double **F_C,double *t,int i)
{
   F_H[0] = F_C[0][i];
   F_H[1] = F_C[1][i];
   *t = F_C[2][i];
}

void store_rom(double F_H[2], double t, int check)
{
   string txtl = "check_rom/dat_";
   stringstream s;
   s << check;
   txtl += s.str() + ".txt";
   char *txt = new char [txtl.size()+1];
   strcpy (txt, txtl.c_str());
   ofstream cp_rom;
   cp_rom.open(txt);
   cp_rom <<  F_H[0] << " " << F_H[1] << " "  << t;
   cp_rom.close();
}

void restore_rom(double F_H[2],double *t,int check)
{
```

```
   string txtl = "check_rom/dat_";
   stringstream s;
   s << check;
   txtl += s.str() + ".txt";
   char *txt = new char [txtl.size()+1];
   strcpy (txt, txtl.c_str());
   ifstream cp_rom;
   cp_rom.open(txt);
   cp_rom >> F_H[0];
   cp_rom >> F_H[1];
   cp_rom >> *t;
   cp_rom.close();
}

double func_U(double t)
{
return 2.*(pow(e,3.*t)-pow(e,3))/(pow(e,3.*t/2.)*(2.+pow(e,3)));
}

void func(double X[2],double t, double F[2])
{
   F[0] = 0.5*X[0]+ func_U(t);
   F[1] = X[0]*X[0]+0.5*(func_U(t)*func_U(t));
}

void func_adj(double BF[2], double X[2], double BX[2])
{
   BX[0] = 0.5*BF[0]+2.*X[0]*BF[1];
   BX[1] = 0.;
}
```

References

[1] R. Becker and R. Rannacher. An optimal control approach to error estimation and mesh adaptation in finite element methods. In *Acta Numerica 2000*, pages 1–101. Iserles A (ed). Cambridge University Press, 2001.

[2] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, pages 35–54, 1992.

[3] A. Griewank and A. Walther. Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Software*, 26:19 – 45, 2000.

[4] M.D. Gunzburger. *Perspectives in flow control and optimization.* Advances in Design and Control 5. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM). xiv, 261 p. , 2003.

[5] P. Stumm and A. Walther. Multistage approaches for optimal offline checkpointing. *SIAM Journal on Scientific Computing*, 31(3):1946–1967, 2009.

[6] P. Stumm and A. Walther. New algorithms for optimal online checkpointing. *SIAM Journal on Scientific Computing*, 32(2):836–854, 2010.

[7] A. Walther. Automatic differentiation of explicit Runge-Kutta methods for optimal control. *Comput. Optim. Appl.*, 36(1):83–108, 2007.

[8] Q. Wang, P. Moin, and G. Iaccarino. Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation. *SIAM Journal on Scientific Computing*, 31(4):2549–2567, 2009.