Struct Multidisc Optim (2010) 41:453–464
DOI 10.1007/s00158-009-0430-0

EDUCATIONAL ARTICLE

A discrete level-set topology optimization code written in Matlab

Vivien J. Challis

Received: 26 November 2008 / Revised: 7 August 2009 / Accepted: 8 August 2009 / Published online: 25 September 2009

Abstract This paper presents a compact Matlab implementation of the level-set method for topology optimization. The code can be used to minimize the compliance of a statically loaded structure. Simple code modifications to extend the code for different and multiple load cases are given. The code is inspired by a Matlab implementation of the solid isotropic material with penalization (SIMP) method for topology optimization (Sigmund, Struct Multidiscipl Optim 21:120–127, 2001). Including the finite element solver and comments, the code is 129 lines long. The code is intended for educational purposes, and in particular it could be used alongside the Matlab implementation of the SIMP method for topology optimization to demonstrate the similarities and differences between the two approaches.

Keywords Topology optimization · Level-set method · Matlab code · Education

1 Introduction

This paper presents a simple Matlab implementation of the level-set method for topology optimization applied to compliance minimization of statically loaded structures. The code is intended for educational purposes and can be downloaded as supplementary material available with this article. This article is inspired by a similar one (Sigmund 2001) which presented a simple Matlab implementation of the solid isotropic material with penalization (SIMP) method for topology optimization.

Level-set methods for moving interface problems were first developed by Osher and Sethian (1988); the methods are by now well-developed and used in a wide variety of research areas (see, e.g., Sethian 1999; Osher and Fedkiw 2002). The level-set method was first applied to topology optimization only recently (Sethian and Wiegmann 2000; Osher and Santosa 2001; Wang et al. 2003; Allaire et al. 2004) and a number of research groups around the world have used the approach successfully as an alternative to the traditional material distribution approach for topology optimization (see the recent monograph by Bendsøe and Sigmund (2004) for an overview of the traditional approach). Recent examples of success with level-set based methods for topology optimization include optimization of the robust compliance (de Gournay et al. 2008) and design of multifunctional materials (Challis et al. 2008a).

The Matlab code provides insight into the main facets of the level-set algorithm for topology optimization and in particular provides concrete implementation details. This will be useful for those new to field of topology optimization as well as those who have only used other topology optimization approaches. Unlike some other implementations of the level-set method (e.g., Allaire et al. 2004), the author's implementation avoids the use of intermediate densities. Hence clear boundaries of the structure exist at all iterations of the algorithm. The author and collaborators have found this particularly useful to allow optimization for

Electronic supplementary material The online of this article (doi:10.1007/s00158-009-0430-0) contains supplementary material, which is available to authorized users.

V. J. Challis (✉)
Department of Mathematics, The University of Queensland, Brisbane, Queensland 4072, Australia
e-mail: vchallis@maths.uq.edu.au

fracture toughness (Challis et al. 2008b) and in the setting of fluid flow (Challis and Guest 2009).

To the author's knowledge, two other level-set topology optimization codes are freely available. One is available at http://www.cmap.polytechnique.fr/~allaire/freefem_en.html. It uses the free software toolbox FreeFem++ and is an extension of the work of Allaire and Pantz (2006). The other level-set code is written in Matlab and is available at http://www2.acae.cuhk.edu.hk/~cmdl/download.htm. However, the code in this article is simpler and shorter, having been particularly designed with an educational setting in mind.

The Matlab implementation presented here includes major aspects of the level-set algorithm in a simple way, and is accompanied by the detailed descriptions in this paper. Sigmund's (2001) Matlab implementation of the SIMP algorithm was 99 lines long. The code presented here is also very compact, and is 129 lines in total. Excluding the finite element code and comments, the level-set topology optimization code is only 63 lines.

The remainder of the paper is as follows. The topology optimization problem and algorithm are outlined in Section 2. The code is explained in Section 3. Extensions to the code for different and multiple load cases are given in Section 4. The discussion in Section 5 addresses how to choose the various parameters of the algorithm and describes other possible extensions. Conclusions are drawn in Section 6. The code itself is given in the Appendix.

2 Topology optimization algorithm

The optimization problem is to minimize the compliance of a solid structure subject to a constraint on the volume of material used. The following is the mathematical description of the problem:

$$\begin{aligned} \min_{\mathbf{x}} : c(\mathbf{x}) &= \mathbf{U}^{\mathrm{T}}\mathbf{K}\mathbf{U} = \sum_{e=1}^{N} \mathbf{u}_e^{\mathrm{T}}\mathbf{k}_e\mathbf{u}_e = \sum_{e=1}^{N} x_e\mathbf{u}_e^{\mathrm{T}}\mathbf{k}_1\mathbf{u}_e \\ \text{subject to} : V(\mathbf{x}) &= V_{req} \\ &: \mathbf{K}\mathbf{U} = \mathbf{F} \\ &: x_e = 0 \text{ or } x_e = 1 \quad \forall\, e = 1, \ldots, N \end{aligned} \tag{1}$$

where $\mathbf{x} = (x_1, \ldots, x_N)$ is the vector of element "densities", with entries of $x_e = 0$ for a void element and $x_e = 1$ for a solid element, where e is the element index. $c(\mathbf{x})$ is the compliance objective function. $\mathbf{F}$ and $\mathbf{U}$ are the global force and displacement vectors, respectively, and $\mathbf{K}$ is the global stiffness matrix. The equation $\mathbf{KU} = \mathbf{F}$ is therefore the elasticity equations as discretized with the finite element method. $\mathbf{u_e}$ is the element displacement vector and $\mathbf{k}_e$ is the element stiffness matrix for element e. $\mathbf{k}_1$ is the element stiffness matrix corresponding to a solid element. N is the total number of elements in the design domain, $V(\mathbf{x})$ is the number of solid elements and V_{req} is the required number of solid elements.

The level-set method (Allaire et al. 2004; Wang et al. 2003) is used to find a local minimum for the optimization problem. The approach derives its name from the use of a level-set function for describing the structure. If the structure occupies some domain Ω, the level-set function has the following property:

$$\psi(\mathbf{p}) \begin{cases} < 0 & \text{if } \mathbf{p} \in \Omega, \\ = 0 & \text{if } \mathbf{p} \in \partial\Omega, \\ > 0 & \text{if } \mathbf{p} \notin \Omega, \end{cases} \tag{2}$$

where $\mathbf{p}$ is any point in the design domain, and $\partial\Omega$ is the boundary of Ω.

The following evolution equation is used to update the level-set function and hence the structure:

$$\frac{\partial\psi}{\partial t} = -v|\nabla\psi| - \omega g, \tag{3}$$

where t represents time, $v(\mathbf{p})$ and $g(\mathbf{p})$ are scalar fields over the design domain, and ω is a positive parameter which determines the influence of the term involving g. The field v determines geometric motion of the boundary of the structure and is chosen based on the shape derivative of the optimization objective. The term involving g is a forcing term which determines the nucleation of new holes within the structure and is chosen based on the topological derivative of the optimization objective.

If the parameter ω is zero, (3) is the standard Hamilton-Jacobi evolution equation for a level-set function ψ under a normal velocity of the boundary $v(\mathbf{p})$, taking the boundary normal in the outward direction from Ω (e.g., Sethian 1999; Osher and Fedkiw 2002). The simpler equation without the term involving g is typically used in level-set methods for topology optimization (e.g., Allaire et al. 2004; Wang et al. 2003). However, on two dimensional problems this standard evolution equation has the major drawback that new void regions cannot be nucleated within the structure (Allaire et al. 2004). Here, the additional forcing term involving g is added following Burger et al. (2004) to ensure that new holes can nucleate within the structure during the optimization process.

To build a topology optimization algorithm, the level-set function can be discretized with gridpoints centered on the elements of the mesh. If $\mathbf{c}_e$ represents

the position of the center of element e, then the discretized level-set function Ψ satisfies:

$$\Psi(\mathbf{c}_e) \begin{cases} < 0 & \text{if } x_e = 1, \\ > 0 & \text{if } x_e = 0. \end{cases} \tag{4}$$

The discrete level-set function can then be updated to find a new structure by solving (3) numerically. Ψ is initialized as a signed distance function and an upwind finite difference scheme is used so that the evolution equation can be accurately solved. In addition, the time step for the finite difference scheme is chosen to satisfy the CFL stability condition

$$\Delta t \leq \frac{h}{\max |v|}, \tag{5}$$

where h is the minimum distance between adjacent gridpoints in the spatial discretization and the maximum is taken over all gridpoints (e.g., Sethian 1999; Osher and Fedkiw 2002).

As stated above, the two scalar fields v and g are typically chosen based on the shape and topological sensitivities of the optimization objective, respectively. In order to satisfy the volume constraint, here they are chosen using the shape and topological sensitivities of the Lagrangian

$$L = c(\mathbf{x}) + \lambda^k \left(V(\mathbf{x}) - V_{req}\right) + \frac{1}{2\Lambda^k}\left[V(\mathbf{x}) - V_{req}\right]^2, \tag{6}$$

where λ^k and Λ^k are parameters which change with each iteration k of the optimization algorithm. They are updated using the scheme

$$\lambda^{k+1} = \lambda^k + \frac{1}{\Lambda^k}\left(V(\mathbf{x}) - V_{req}\right), \qquad \Lambda^{k+1} = \alpha\Lambda^k, \tag{7}$$

where $\alpha \in (0, 1)$ is a fixed parameter. This implements the augmented Lagrangian multiplier method for constrained optimization, as used for topology optimization with the level-set method by Luo et al. (2008a).

The normal velocity v is chosen as a descent direction for the Lagrangian L using its shape derivative (e.g., Allaire et al. 2004; Wang et al. 2003). In the case of traction-free boundary conditions on the moving boundary, the shape sensitivity of the compliance objective $c(\mathbf{x})$ is the negative of the strain energy density: (e.g., Allaire et al. 2004)

$$\frac{\delta c}{\delta \Omega}|_e = -\mathbf{u}_e^{\mathrm{T}}\mathbf{k}_e\mathbf{u}_e. \tag{8}$$

The shape sensitivity of the volume $V(\mathbf{x})$ is

$$\frac{\delta V}{\delta \Omega}|_e = 1. \tag{9}$$

Using these shape sensitivities, the normal velocity v within element e at iteration k of the algorithm is

$$v|_e = -\frac{\delta L}{\delta \Omega}|_e = \mathbf{u}_e^{\mathrm{T}}\mathbf{k}_e\mathbf{u}_e - \lambda^k - \frac{1}{\Lambda^k}\left(V(\mathbf{x}) - V_{req}\right). \tag{10}$$

Based on Burger et al. (2004), the forcing term g should be taken as

$$g = -\mathrm{sign}(\psi)\delta_{\mathrm{T}}L, \tag{11}$$

where $\delta_{\mathrm{T}}L$ is the topological sensitivity of the Lagrangian L. For compliance minimization, nucleating solid areas within the void regions of the design is pointless because such solid regions will not take any load. Therefore holes should only be nucleated within the solid structure and

$$g = \begin{cases} \delta_{\mathrm{T}}L & \text{if } \psi < 0 \\ 0 & \text{if } \psi \geq 0. \end{cases} \tag{12}$$

The topological sensitivity of the compliance objective function in two dimensions with traction-free boundary conditions on the nucleated hole and the unit ball as the model hole is (e.g., Allaire et al. 2005)

$$\delta_{\mathrm{T}}c|_e = \frac{\pi(\lambda + 2\mu)}{2\mu(\lambda + \mu)}\left(4\mu\mathbf{u}_e^{\mathrm{T}}\mathbf{k}_e\mathbf{u}_e + (\lambda - \mu)\mathbf{u}_e^{\mathrm{T}}(\mathbf{k_{Tr}})_e\mathbf{u}_e\right) \tag{13}$$

where $\mathbf{u}_e^{\mathrm{T}}(\mathbf{k_{Tr}})_e\mathbf{u}_e$ is the finite element approximation to the product $\mathrm{tr}(\sigma)\mathrm{tr}(\epsilon)$ where σ is the stress tensor and ϵ is the strain tensor. In (13), λ and μ are the Lamé constants for the solid material. The topological sensitivity of the volume $V(\mathbf{x})$ when the model hole is the unit ball is

$$\delta_{\mathrm{T}}V|_e = -\pi. \tag{14}$$

Putting these results into the definition of the Lagrangian and using (12) gives the source term g.

Those readers interested in more discussion of the algorithm are referred to the publications referenced above. Also, many more details are revealed by the Matlab implementation and are discussed in the following section.

3 Explanation of Matlab implementation

Following Sigmund (2001), several assumptions are made to simplify the Matlab code. In particular, the optimization domain is assumed to be rectangular and split into square finite elements. There are `nelx` elements along the horizontal direction and `nely`

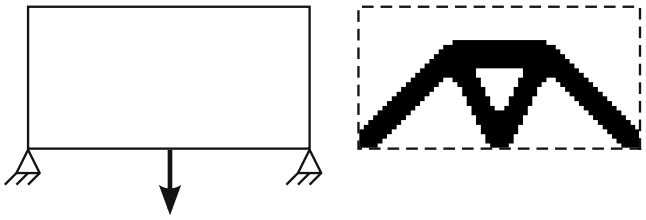

Fig. 1 *Left*: load and supports for the simple bridge problem, and *right*: the result of `top_levelset(60,30,0.3, 3,2,2);`

elements along the vertical direction.[1] The total number of elements is $N = \texttt{nelx} \times \texttt{nely}$.

The code is envoked with the call:

```
top_levelset(nelx,nely,volReq,
  stepLength,numReinit,topWeight);
```

where `volReq` is the required solid fraction (equivalent to $\frac{V_{req}}{N}$), and the remaining three inputs are parameters of the algorithm. The parameter `stepLength` specifies the time interval over which the level-set function is evolved in a design update, measured relative to the CFL time step. The parameter `numReinit` determines how often the level-set function is reinitialized to a signed distance function: level-set reinitialization is performed at the end of each `numReinit`-th iteration of the algorithm. The last parameter, `topWeight`, is the weighting ω of the forcing term in the evolution equation.

The boundary conditions and loads are specified within the Matlab code. The version of the code given here performs the optimization for a simple bridge. A suggested first call of the program is:

```
top_levelset(60,30,0.3,3,2,2);
```

The boundary conditions and optimization result for this call to the code are given in Fig. 1.

Details of the code are explained in the following subsections. Wherever possible the code is similar to that of Sigmund (2001).

3.1 Main program—lines 1–48

The main program includes initialization, an iteration loop to perform the optimization and a check for convergence.

The initialization phase defines the array `struc` of size $\texttt{nely} \times \texttt{nelx}$ that stores the current values x_e defining the structure. The initial structure is set to be entirely solid (line 4). The level-set function `lsf` is initialized from the structure using the `reinit` function (line 5) which is described in detail in Section 3.2. The shape and topological sensitivity arrays `shapeSens` and `topSens` are initialized as zero arrays (line 6). Lastly, the stiffness matrix and other material parameters are set with a call to `materialInfo` (line 7) before iteration of the algorithm begins.

Iterations are counted with `iterNum` and continue for a maximum of 200 iterations (line 9). Inside the iteration loop, the finite element analysis routine `FE` is called (line 11) to allow calculation of the shape and topological sensitivities for the current structure (lines 12–21). The node numbering and method of finding `Ue` ($= \mathbf{u}_e$) in lines 12–16 are drawn directly from Sigmund (2001). Line 17 calculates the shape sensitivity of the compliance from (8).[2] Lines 18–19 calculate the topological sensitivity of the compliance from (13). The compliance objective value is calculated from the shape sensitivity and stored in the vector `objective` (line 23), and the current solid volume fraction is calculated as `volCurr` (line 24). The objective and volume information for the current structure is displayed to the screen and the current structure is displayed as a figure (lines 25–27), similar to Sigmund (2001).

A convergence check (lines 28–32) may terminate the algorithm before 200 iterations are completed. The convergence check is not performed for the first five iterations of the algorithm (line 29). After these first five iterations, the optimization terminates if the volume is within 0.005 of the required value `volReq` (line 29) and also the previous five objective function values are all within a 1% tolerance of the current objective value (line 30).

If the algorithm does not terminate, a design update is performed. Prior to the design update, the parameters for imposing the volume constraint with the augmented Lagrangian method are updated (lines 33–38). The Matlab variable `la` stores the current value of λ^k, `La` stores the current value of Λ^k, and `alpha` stores the parameter α. The sensitivities of the additional terms in the Lagrangian in (6) are added to the sensitivities of the objective function (lines 39–41) and then the design update is performed with a call to `updateStep` (line 43). The reinitialization of the level-set function is then carried out, but only for iteration numbers that are a multiple of the user-specified parameter `numReinit` (line 44–47). The optimization loop then repeats.

[1] Names in typewriter font correspond to Matlab variables or functions.

[2] The factor `max(struc(ely,elx),0.0001)` gives a very small stiffness to the void phase which improves the robustness of the algorithm.

3.2 Reinitialization of the level-set function—lines 49–53

The reinitialization of the level-set function to a signed distance function is important to ensure accuracy in solving the evolution equation (e.g., Sethian 1999; Osher and Fedkiw 2002). However, to allow nucleation of holes under evolution of (3), the level-set function should not be reinitialized too often (Burger et al. 2004). This is controlled with the user-specified parameter `numReinit`. When required, reinitialization is achieved in the Matlab implementation using the function `reinit`, which accepts the current structure `struc` and returns the level-set function `lsf` ($= \Psi$) for that structure (line 50).

The level-set array `lsf` has size `(nely+2)` $\times$ `(nelx+2)`—this includes a border of void elements around the structure. The border is necessary to ensure the boundary of the structure is accurately represented by the level-set function. `struc` is therefore enlarged to `strucFull` which has a border of void elements (line 51) prior to calculating `lsf`.

For each element, the level-set function is calculated as the minimum Euclidean distance to an element of the opposite phase, as measured from the center of the element. For solid elements the sign is chosen to be negative, whereas for void elements the sign is chosen to be positive, matching with the definition of the level-set function (4).

Matlab's `bwdist` function (part of the Image Processing Toolbox) makes the reinitialization extremely simple. Given an array, `bwdist` returns an array of the same size where each entry is the distance to the nearest non-zero entry of the array. With careful use of signs and logical expressions, two calls to `bwdist` generates the level-set function (line 53).

For readers without access to Matlab's Image Processing Toolbox, the author has written a simple version of `bwdist`. The code is given in Appendix B.

3.3 Design update—lines 54–63

The function `updateStep` implements a design update using the shape and topological sensitivity information. The implementation involves the following steps:

1. Smooth the sensitivities (lines 56–58) to avoid very sharp changes between neighbouring elements (such as across the boundary of the structure). Including this heuristic step is not necessary but tends to give better results. The Matlab function `padarray` which is used in this smoothing step is part of the Image Processing Toolbox. A replacement function for those readers without the toolbox is given in Appendix B.
2. Set the sensitivities to zero for those elements which must not change phase during the evolution. In particular, any load-bearing elements must remain solid (lines 59–61). If desired, passive elements can also be specified to remain void by setting their sensitivities to zero.
3. Evolve the level-set function to find the new structure using the function `evolve` (lines 62–63). The negative of the shape sensitivity `shapeSens` is passed into the `evolve` function as the normal velocity v for the boundary of the design. The expression `topSens.*(lsf(2:end-1,2:end-1)<0)` in line 63 calculates the forcing term g from the topological sensitivity of the Lagrangian `topSens` as in (12). The user-specified parameter `topWeight` is passed into `evolve` as the parameter ω. The `evolve` function is described next.

3.4 Evolution of the level-set function—lines 64–86

The function `evolve` solves the evolution equation (3) to give a new structure. The function takes in the current level-set function `lsf` ($= \Psi$), the normal velocity `v` ($= v$), the forcing term `g` ($= g$), the user-specified parameter `stepLength`, and the parameter `w` ($= \omega$).

This function uses an upwind finite difference scheme (e.g., Sethian 1999, page 65) to solve the evolution equation (3). The time-step is chosen to be a small portion (0.1) of the value given by the CFL condition in (5) (lines 69–70), and the number of time-steps is chosen based on the user specified value `stepLength` (lines 71–72).

The finite difference scheme is fairly simple to implement using Matlab's `circshift` function. The velocity `v` and source term `g` are extended with a border of zeros on lines 66–68 to match the level-set function which includes a border of void elements around the edge of the domain. The new structure is found from the sign of the updated level-set function (lines 85–86).

3.5 Finite element implementation—lines 87–106

The function `FE` which implements the finite element method is almost identical to the implmentation by Sigmund (2001), to which the reader is referred for more information.

The only differences occur in the application of the loads and supports, which are specific to the bridge problem (lines 100–102), and in the assembly of the global stiffness matrix because there is no need to penalize intermediate densities. The factor of `max(struc`

`(elx,ely),0.0001)` in the assembly of the global stiffness matrix (line 97) is to ensure it is non-singular, allowing solution of the system with Matlab's matrix divide operator.[3]

3.6 Element stiffness matrix—lines 107–129

The `materialInfo` function returns the element stiffness matrix `KE` ($= \mathbf{k}_e$), the element matrix for calculating the trace of the stress tensor multiplied by the trace of the strain tensor `KTr` ($= (\mathbf{k_{Tr}})_e$), and the Lamé constants `lambda` ($= \lambda$) and `mu` ($= \mu$) (line 108). The calculation of the element stiffness matrix is the same as the implementation given by Sigmund (2001), but is rearranged to allow re-use of the code for calculating the matrix `KTr`.

4 Simple extensions

It is easy to alter the code to consider different boundary conditions and loads, multiple load cases, keeping certain areas void or solid, and different initial structures. Examples of the first two extensions are given here.[4]

4.1 Other boundary conditions

It is straightforward to change the code to solve different optimization problems. Two examples are presented here.

First, we change the boundary condition at the lower right-hand corner of the bridge example of Fig. 1 to consider the half-wheel design problem shown in Fig. 2. To restrict only the vertical motion of the gridpoint at the lower right-hand corner of the design domain, line 102 in the code is replaced with:

```
102 fixeddofs = [(2*(nely+1)-1):(2*(nely+1))
      ,2*(nelx+1)*(nely+1)];
```

The comment in line 100 should also be changed to reflect the new boundary conditions and be replaced with:

```
100 %Define loads and supports - Half-wheel:
```

[3]It is possible to ascribe zero stiffness to the void phase, and then solve the equation $\mathbf{KU} = \mathbf{F}$ via a conjugate gradient algorithm. As noted earlier, a small nonzero stiffness for the void phase improves the robustness of the algorithm.

[4]Readers should be careful if copying and pasting code snippets from this article into Matlab. The author has found that spaces are sometimes added during this process to give incorrect Matlab code.

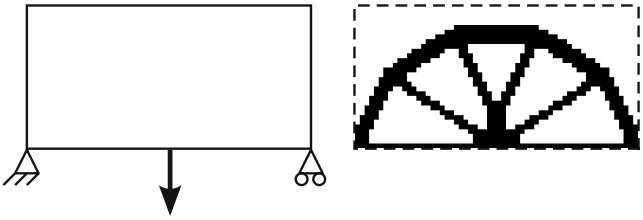

Fig. 2 *Left*: the load and supports for the half-wheel problem, and *right*: the result of `top_levelset(60,30,0.3,3,2,4);` with the code altered to optimize this half-wheel

Note that the load for the half-wheel is the same as the bridge example so does not need to be changed.

With these new lines, the half-wheel optimization can be called with (for example):

```
top_levelset(60,30,0.3,3,2,4);
```

The final structure is displayed in Fig. 2.

Next, we consider a simple cantilever problem depicted in Fig. 3, matching Sigmund (2001). For this example, the code must be changed in two places because both the boundary conditions and the placement of the load have changed. The load bearing elements for the canitlever must be set to remain solid during the optimization. This is done by setting the sensitivities `shapeSens` and `topSens` of the elements to zero within the `updateStep` function. Specifically, lines 59–61 are replaced with:

```
59 % Load bearing elements must remain
     solid - Cantilever:
   shapeSens(end,end) = 0;
   topSens(end,end) = 0;
```

In addition, the applied force and supports must be set within the finite element implementation `FE`, replacing lines 100–102:

```
100 %Define loads and supports - Cantilever:
    F((nelx+1)*(nely+1)*2,1) = 1;
    fixeddofs    = 1:2*(nely+1);
```

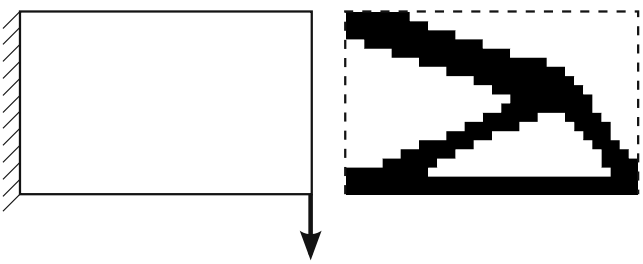

Fig. 3 *Left*: the load and supports for the cantilever problem, and *right*: the result of `top_levelset(32,20,0.4,2,3,2);` with the code altered to optimize this cantilever

With these new lines, the cantilever optimization can be called with (for example):

```
top_levelset(32,20,0.4,2,3,2);
```

The final structure is displayed in Fig. 3.

4.2 Multiple load cases

Allowing for multiple load cases is also not difficult, and the code can be extended for this purpose in a very similar fashion to the Matlab SIMP implementation (Sigmund 2001). Here horizontal loads are added to the bridge problem to improve the stability of the bridge. The new loads are depicted in Fig. 4. The new objective is the sum of the compliance from the three load cases.

The following line is added after line 11 to set the shape and topological sensitivities to zero before adding in the sensitivites from each load case:

```
11a shapeSens(:) = 0; topSens(:) = 0;
```

To accumulate `shapeSens` and `topSens` from the three load cases, lines 16–19 are replaced with the following loop:

```
16 for i=1:3
   Ue = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2
     +1;2*n2+2; 2*n1+1;2*n1+2],i);
   shapeSens(ely,elx) = shapeSens(ely,elx)
     - max(struc(ely,elx),0.0001)*Ue'*KE*Ue;
   topSens(ely,elx) = topSens(ely,elx) +
     struc(ely,elx)*pi/2* ...
     (lambda+2*mu)/mu/(lambda+mu)*(4*mu*Ue
       '*KE*Ue+(lambda-mu)*Ue'*KTr*Ue);
   end
```

Finally, changes are required in the finite element section of the code to account for now having three load cases. Line 91 becomes:

```
91 F = sparse(2*(nely+1)*(nelx+1),3); U =
     zeros(2*(nely+1)*(nelx+1),3);
```

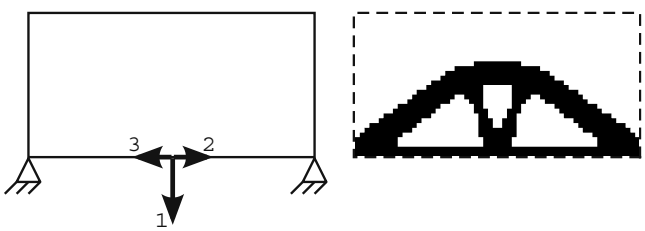

Fig. 4 *Left*: the loads and supports for the multiple load bridge problem—the three loads are applied separately as indicated by the numbers 1, 2 and 3, and *right*: the result of `top_levelset(60,30,0.3,4,2,3);`, with the code changed to consider this multi-load bridge problem

The two additional loads must be specified after line 101 as:

```
101a F(2*(round(nelx/2)+1)*(nely+1)-1,2) =
       0.5;
     F(2*(round(nelx/2)+1)*(nely+1)-1,3) =
       -0.5;
```

Figure 4 is the result of this multi-load version of the code called with the following:

```
top_levelset(60,30,0.3,4,2,3);
```

The bridge optimized with the three loads applied separately is obviously more stable under horizontal loads than the one optimized with the three loads applied together as in Fig. 1.

5 Discussion

This discussion addresses the various parameters of the Matlab implementation of the level-set algorithm. It also describes some additional extensions that might be useful for tackling more complicated topology optimization problems.

A table of suggested parameter values for the six parameters which are specified in the call to `top_levelset` are given in Table 1. The upper and lower bounds for the parameters have been determined by experimenting with the Matlab code on various simple compliance minimization problems. Choosing different parameters within the suggested ranges can result in different topology optimization results. However, using values within the suggested ranges tend to give sensible optimization results that have objective values close to optimal objective value. Exceptions can occur, and in particular the two parameters `numReinit` and `topWeight` both affect the ability of the algorithm to nucleate holes in the design so need to be chosen carefully. Specifically, choosing both parameters at the lower end of the suggested ranges (`numReinit` $= 2$ and `topWeight` $= 1$) can mean that new holes cannot be nucleated during the optimization and give poor optimized designs.

Other parameters are hard-coded into the Matlab code. It is not necessary to complicate the program call by including them because the hard-coded values have been suitable for all of the optimization problems tested by the author. The meanings of these parameters along with any relevant comments are given in Table 2.

As well as altering parameters to see how this affects the outcome of the optimization, readers may wish to implement new features into the Matlab code. Based

Table 1 Suggested parameter values for the six parameters in the program call `top_levelset(nelx,nely,volReq,stepLength,numReinit,topWeight);`

Parameter	Meaning	Suggested range of values	If too small, then:	If too large, then:
`nelx`	Number of elements along the width of the design domain	Integers greater than 20 with `nelx` × `nely` < 5000	Poor finite-element representation of design domain	Optimization takes a long time to run
`nely`	Number of elements along the height of the design domain			
`volReq`	Required volume fraction of the final design	Must be between 0 and 1, typically between 0.2 and 0.7	Not enough solid elements to give a sensible design	Almost the entire design domain will be solid
`stepLength`	Number of CFL time steps for which the evolution equation is solved at each iteration of the algorithm	Integer value between `min(nelx,nely)/10` and `max(nelx,nely)/5`	The design will change too slowly and converge to a poor local minimum	The design will change too quickly (possibly removing important features) and converge to a poor local minimum
`numReinit`	Number of iterations of the algorithm before a level-set reinitalization is performed	Integer value between 2 and 6	Reinitialization is performed so often that no new holes can be nucleated in the design	The level-set function becomes very steep, leading to poor accuracy when solving the evolution equation
`topWeight`	Weighting of the topological derivative term in the evolution equation	Any real number between 1 and 4	No holes can be nucleated in the design because the shape derivative dominates the design evolution	The topological derivative dominates the evolution to generate large holes with jagged edges

on the author's experience, there are three extensions which particularly come to mind:

- Improving robustness of the algorithm by checking the change in the objective at each iteration.
- Extending the code in order to impose multiple constraints.
- Utilizing symmetry to speed up the finite element calculations during the optimization process.

A brief description of how to implement these features follows.

Checking the change in the objective prior to accepting an iteration of the optimization algorithm is straight-forward to implement but damages the simple iteration loop of the presented Matlab code so has not been presented here by the author. If the objective increases too much after the design has been updated with a call to `updateStep`, the new structure can be rejected and the author would suggest an adaptive correction of the parameter `stepLength` prior to a new call to `updateStep`. By reducing `stepLength` to a fraction of its previous value, the new call to `updateStep` will result in less change in the design. Note that it is necessary for some increase in the objective function to be acceptable, therefore the tolerance on this checking process should not be set too tightly. The reason for this is two-fold: some ability to go up-hill can help escape local minima, and, when an entirely solid initial design is used the optimization will inevitably go up-hill.

To tackle more complicated optimization problems, it is important to be able to deal with more than one constraint in the optimization problem. The augmented Lagrangian technique applied in the Matlab code to implement the volume constraint can be used with more than one constraint. For example, the technique has been used with two constraints in conjunction with the level-set method of topology optimization by Luo et al. (2008b). The author and colleagues have developed an alternative approach, which was successful for imposing an isotropy constraint when designing materials in both two and three dimensions. The reader is referred to Challis et al. (2008a).

In the Matlab implementation of the SIMP method (Sigmund 2001), it is easy to utilize symmetry to reduce the size of the design domain and improve the speed of the finite element calculations. This only requires implementation of the correct boundary conditions along the line of symmetry. To utilize symmetry with in the level-set approach, both the reinitialization of the level-set function and the finite difference evolution scheme must be modified. This is necessary so that the level-set function accurately reflects the symmetry of the optimization problem. The code changes can be made

Table 2 Description of parameters hard-coded into the Matlab code

Parameter	Meaning and relevant comments
`ones(nely,nelx)` on line 4	Initial structure from which to begin the optimization.
200 on line 9	Maximum number of iterations of the optimization algorithm.
0.0001 on line 17	Stiffness of the void phase compared to the solid phase. This can be zero but a small nonzero value improves robustness of the algorithm.
5 on line 29	Number of iterations at the start of the optimization for which the convergence criteria are not checked. This is included so that the condition on line 30 is only checked when it is valid (otherwise the array index `end-5` does not make sense).
0.005 on line 29	Tolerance for satisfaction of the volume constraint.
5 on line 30	Number of iterations which must have similar objective values for termination of the algorithm. If changed, the user must also change the 5 on line 29.
0.01 on line 30	Relative tolerance on the objective values for termination of the algorithm.
-0.01, 1000 and 0.9 on line 35	Initial parameters for imposing the volume constraint with the augmented Lagrangian method. If the initial value of `La` is too small or the value of `alpha` is too small then the optimization can become unstable.
array on lines 57 and 58	Convolution filter to smooth the sensitivities, currently chosen as `1/6*[0 1 0;1 2 1;0 1 0]`.
0.1 on line 70	Fraction of the CFL time step to use as a time step for solving the evolution equation. This must be less than or equal to 1 to satisfy the CFL condition. Also, to ensure the correct interpretation of the parameter `stepLength`, this value must be the reciprocal of the parameter 10 on line 72. If that value is changed, this must be changed to match.
10 on line 72	Number of time steps required to evolve the level-set function for a time equal to the CFL time step. To ensure the correct interpretation of the parameter `stepLength`, this value must be the reciprocal of the parameter 0.1 on line 70. If this value is changed, that must be changed to match.
0.0001 on line 97	Stiffness of the void phase compared to the solid phase. This must be a positive nonzero value so that the global stiffness matrix is not singular, and should match the parameter 0.0001 on line 17.
1 on line 110	The Young's modulus of the solid phase.
0.3 on line 110	The Poisson's ratio of the solid phase.

with care on a case-by-case basis. These changes are also required when designing unit cells for material microstructures, so that periodicity of the design is taken into account.

6 Conclusion

This paper presents a simple Matlab implementation of the level-set method for topology optimization. The code can be used to demonstrate the capabilities of the algorithm as well as specific implementation details.

The code could easily be used alongside the SIMP Matlab code written by Sigmund (2001) for simple comparisons of the two methods in an educational setting. Despite having more lines than the SIMP implementation the level-set Matlab code is still very compact, and neglecting comments and the finite element code is only 63 lines long.

Acknowledgements The author would like to acknowledge Prof. M. P. Bendsøe, Prof. J. K. Guest, Dr. A. P. Roberts, Prof. O. Sigmund and Dr. A. H. Wilkins for their helpful discussions and support of this work. The author would also like to thank the anonymous reviewers for their useful comments that resulted in significant improvements to the manuscript. The author has received an Australian Postgraduate Award and funding from the Australian Research Council (grant DP0878785). This financial support is gratefully acknowledged.

Appendix A—Matlab code

```
1  %% TOPOLOGY OPTIMIZATION USING THE LEVEL-
       SET METHOD, VIVIEN J. CHALLIS 2009
2  function [struc] = top_levelset(nelx,
     nely,volReq,stepLength,numReinit,
     topWeight)
3  % Initialization
4  struc = ones(nely,nelx);
5  [lsf] = reinit(struc);
6  shapeSens = zeros(nely,nelx); topSens =
     zeros(nely,nelx);
7  [KE,KTr,lambda,mu] = materialInfo();
8  % Main loop:
9  for iterNum = 1:200
10  % FE-analysis, calculate sensitivities
11  [U] = FE(struc,KE);
12  for ely = 1:nely
13   for elx = 1:nelx
14    n1 = (nely+1)*(elx-1)+ely;
```

```
    n2 = (nely+1)* elx   +ely;
    Ue = U([2*n1-1;2*n1; 2*n2-1;2*n2; 2*n2
      +1;2*n2+2; 2*n1+1;2*n1+2],1);
    shapeSens(ely,elx) = -max(struc(ely,
      elx),0.0001)*Ue'*KE*Ue;
    topSens(ely,elx) = struc(ely,elx)*
      pi/2*(lambda+2*mu)/mu/(lambda+mu)*
      ...
      (4*mu*Ue'*KE*Ue+(lambda-mu)*Ue'*KTr*
        Ue);
  end
 end
 % Store data, print & plot information
 objective(iterNum) = -sum(shapeSens(:));
 volCurr = sum(struc(:))/(nelx*nely);
 disp([' It.: ' num2str(iterNum)       '
   Compl.: ' sprintf('%10.4f',objective(
   iterNum)) ...
    ' Vol.: ' sprintf('%6.3f',volCurr)])
 colormap(gray); imagesc(-struc,[-1,0]);
   axis equal; axis tight; axis off;
   drawnow;
 % Check for convergence
 if iterNum > 5 && ( abs(volCurr-volReq)
   < 0.005 ) && ...
   all( abs(objective(end)-objective(
     end-5:end-1) ) < 0.01*abs(objective(
     end)) )
  return;
 end
 % Set augmented Lagrangian parameters
 if iterNum == 1
  la = -0.01; La = 1000; alpha = 0.9;
 else
  la = la - 1/La * (volCurr - volReq); La
     = alpha * La;
 end
 % Include volume sensitivities
 shapeSens = shapeSens - la + 1/La*(
   volCurr-volReq);
 topSens = topSens + pi*(la - 1/La*(
   volCurr-volReq));
 % Design update
 [struc,lsf] = updateStep(lsf,shapeSens,
   topSens,stepLength,topWeight);
 % Reinitialize level-set function
 if ~mod(iterNum,numReinit)
  [lsf] = reinit(struc);
 end
end
%%---- REINITIALIZATION OF LEVEL-SET
   FUNCTION ----
function [lsf] = reinit(struc)
strucFull = zeros(size(struc)+2);
   strucFull(2:end-1,2:end-1) = struc;
% Use "bwdist" (Image Processing Toolbox)
lsf = (~strucFull).*(bwdist(strucFull)
   -0.5) - strucFull.*(bwdist(strucFull-1)
   -0.5);
%%----- DESIGN UPDATE ----
function [struc,lsf] = updateStep(lsf,
   shapeSens,topSens,stepLength,topWeight)
% Smooth the sensitivities
[shapeSens] = conv2(padarray(shapeSens
   ,[1,1],'replicate'),1/6*[0 1 0; 1 2 1;
   0 1 0],'valid');
[topSens] = conv2(padarray(topSens,[1,1],
   'replicate'),1/6*[0 1 0; 1 2 1; 0 1 0],
   'valid');
% Load bearing pixels must remain solid -
   Bridge:
shapeSens(end,[1,round(end/2):round(end
   /2+1),end]) = 0;
topSens(end,[1,round(end/2):round(end
   /2+1),end]) = 0;
% Design update via evolution
[struc,lsf] = evolve(-shapeSens,topSens
   .*(lsf(2:end-1,2:end-1)<0),lsf,
   stepLength,topWeight);
%%---- EVOLUTION OF LEVEL-SET FUNCTION
   ----
function [struc,lsf] = evolve(v,g,lsf,
   stepLength,w)
% Extend sensitivites using a zero border
vFull = zeros(size(v)+2); vFull(2:end
   -1,2:end-1) = v;
gFull = zeros(size(g)+2); gFull(2:end
   -1,2:end-1) = g;
% Choose time step for evolution based on
   CFL value
dt = 0.1/max(abs(v(:)));
% Evolve for total time stepLength * CFL
   value:
for i = 1:(10*stepLength)
 % Calculate derivatives on the grid
 dpx = circshift(lsf,[0,-1])-lsf;
 dmx = lsf - circshift(lsf,[0,1]);
 dpy = circshift(lsf,[-1,0]) - lsf;
 dmy = lsf - circshift(lsf,[1,0]);
 % Update level set function using an
   upwind scheme
 lsf = lsf - dt * min(vFull,0).* ...
   sqrt( min(dmx,0).^2+max(dpx,0).^2+min(
     dmy,0).^2+max(dpy,0).^2 ) ...
   - dt * max(vFull,0) .*...
   sqrt( max(dmx,0).^2+min(dpx,0).^2+max(
     dmy,0).^2+min(dpy,0).^2 )...
   - w*dt*gFull;
end
% New structure obtained from lsf
strucFull = (lsf<0); struc = strucFull(2:
   end-1,2:end-1);
%%---- FINITE ELEMENT ANALYSIS ----
function [U] = FE(struc,KE)
[nely,nelx] = size(struc);
K = sparse(2*(nelx+1)*(nely+1), 2*(nelx
   +1)*(nely+1));
F = sparse(2*(nely+1)*(nelx+1),1); U =
   zeros(2*(nely+1)*(nelx+1),1);
for elx = 1:nelx
```

```
 for ely = 1:nely
  n1 = (nely+1)*(elx-1)+ely;
  n2 = (nely+1)* elx   +ely;
  edof = [2*n1-1; 2*n1; 2*n2-1; 2*n2; 2*
    n2+1; 2*n2+2; 2*n1+1; 2*n1+2];
  K(edof,edof) = K(edof,edof) + max(struc
    (ely,elx),0.0001)*KE;
 end
end
% Define loads and supports - Bridge:
F(2*(round(nelx/2)+1)*(nely+1),1) = 1;
fixeddofs = [2*(nely+1)-1:2*(nely+1),2*(
  nelx+1)*(nely+1)-1:2*(nelx+1)*(nely+1)
  ];
% Solving
alldofs = 1:2*(nely+1)*(nelx+1);
freedofs = setdiff(alldofs,fixeddofs);
U(freedofs,:) = K(freedofs,freedofs) \ F(
  freedofs,:);
%%---- MATERIAL INFORMATION ----
function [KE,KTr,lambda,mu] =
  materialInfo()
% Set material parameters, find Lame
  values
E = 1.; nu = 0.3;
lambda = E*nu/((1+nu)*(1-nu)); mu = E
  /(2*(1+nu));
% Find stiffness matrix "KE"
k =[ 1/2-nu/6   1/8+nu/8 -1/4-nu/12
  -1/8+3*nu/8 ...
    -1/4+nu/12 -1/8-nu/8       nu/6
      1/8-3*nu/8];
KE = E/(1-nu^2)*stiffnessMatrix(k);
% Find "trace" matrix "KTr"
k=[1/3 1/4 -1/3 1/4 -1/6 -1/4 1/6 -1/4];
KTr = E/(1-nu)*stiffnessMatrix(k);
%%---- ELEMENT STIFFNESS MATRIX ----
function [K] = stiffnessMatrix(k)
% Forms stiffness matrix from first row
K=[k(1) k(2) k(3) k(4) k(5) k(6) k(7) k
  (8)
  k(2) k(1) k(8) k(7) k(6) k(5) k(4) k(3)
  k(3) k(8) k(1) k(6) k(7) k(4) k(5) k(2)
  k(4) k(7) k(6) k(1) k(8) k(3) k(2) k(5)
  k(5) k(6) k(7) k(8) k(1) k(2) k(3) k(4)
  k(6) k(5) k(4) k(3) k(2) k(1) k(8) k(7)
  k(7) k(4) k(5) k(2) k(3) k(8) k(1) k(6)
  k(8) k(3) k(2) k(5) k(4) k(7) k(6) k(1)
    ];
```

Appendix B—Additional functions to replace calls to Matlab's Image Processing Toolbox

The `bwdist` and `padarray` calls utilize Matlab's Image Processing Toolbox. A simple version of `bwdist` is given here:

```
%%---- VJC'S VERSION OF "bwdist" ----
function dist = bwdist(image)
% Gives Euclidean distance to the closest
   non-zero entry in "image"
dist=zeros(size(image));
[yinds,xinds]=find(image~=0);
for i=1:size(image,2)
 for j=1:size(image,1)
  dist(j,i)=sqrt(min((yinds-j).^2+(xinds-
    i).^2));
 end
end
```

This code is not optimized for speed and will significantly slow down the iterations of the optimization algorithm for problems with large numbers of elements.

A simple version of `padarray` is also given below. The calls to `padarray` in lines 57 and 58 of the above Matlab code are of the form:

```
padarray(array,[1,1],'replicate')
```

With the below code these should be replaced with simply

```
padarray_vjc(array)
```

where the `padarray_vjc` function is defined below.

```
%% ---- VJC's VERSION OF "padarray" ----
function padded_array = padarray_vjc(
  array)
% The call padarray_vjc(array) is
  equivalent to padarray(array,[1,1],'
  replicate')
padded_array = zeros(size(array)+2);
padded_array(2:end-1,2:end-1) = array;
padded_array(2:end-1,      1) = array(1:
  end,  1);
padded_array(2:end-1,    end) = array(1:
  end,end);
padded_array(      1,  1:end) =
  padded_array(    2,1:end);
padded_array( end,  1:end) = padded_array
  (end-1,1:end);
```

References

Allaire G, Pantz O (2006) Structural optimization with FreeFem++. Struct Multidiscipl Optim 32:173–181

Allaire G, Jouve F, Toader A-M (2004) Structural optimization using sensitivity analysis and a level-set method. J Comput Phys 194:363–393

Allaire G, de Gournay F, Jouve F, Toader A-M (2005) Structural optimization using topological and shape sensitivity via a level set method. Control Cybern 34(1):59–80

Bendsøe MP, Sigmund O (2004) Topology optimization: theory, methods and applications, 2nd edn. Springer, Berlin

Burger M, Hackl B, Ring W (2004) Incorporating topological derivatives into level set methods. J Comput Phys 194:344–362

Challis VJ, Guest JK (2009) Level-set topology optimization of fluids in Stokes flow. Int J Numer Methods Eng 79:1284–1308
Challis VJ, Roberts AP, Wilkins AH (2008a) Design of three dimensional isotropic microstructures for maximized stiffness and conductivity. Int J Solids Struct 45:4130–4146
Challis VJ, Roberts AP, Wilkins AH (2008b) Fracture resistance via topology optimization. Struct Multidiscipl Optim 36:263–271
de Gournay F, Allaire G, Jouve F (2008) Shape and topology optimization of the robust compliance via the level set method. ESAIM COCV 14(1):43–70
Luo J, Luo Z, Chen L, Tong L, Wang MY (2008a) A semi-implicit level set method for structural shape and topology optimization. J Comput Phys 227:5561–5581
Luo J, Luo Z, Chen S, Tong L, Wang MY (2008b) A new level set method for systematic design of hinge-free compliant mechanisms. Comput Methods Appl Mech Eng 198: 318–331
Osher S, Fedkiw R (2002) Level set methods and dynamic implicit surfaces. Applied mathematical sciences, vol 153. Springer, New York
Osher SJ, Santosa F (2001) Level set methods for optimization problems involving geometry and constraints I. Frequencies of a two-density inhomogeneous drum. J Comput Phys 171:272–288
Osher SJ, Sethian JA (1988) Fronts propagating with curvature dependent speed: algorithms based on the Hamilton–Jacobi formulation. J Comput Phys 79:12–49
Sethian JA (1999) Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision and materials science. Cambridge Monographs on Applied and Computational Mathematics, vol 3, 2nd edn. Cambridge University Press, Cambridge
Sethian JA, Wiegmann A (2000) Structural boundary design via level set and immersed interface methods. J Comput Phys 163(2):489–528
Sigmund O (2001) A 99 line topology optimization code written in Matlab. Struct Multidiscipl Optim 21:120–127
Wang MY, Wang X, Guo D (2003) A level set method for structural topology optimization. Comput Methods Appl Mech Eng 192:227–246